Adam Zalepa

BLITZ BASIC

Programming base

Łódź 2017

ISBN 978-83-947369-6-5

Publisher:
A2

The real act of discovery
is not about finding new lands,
but about looking at the old
in a new way.

- Marcel Proust

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

INTRODUCTION

This publication is the second book in the series "Programming from scratch". The previous item deals with the language "AMOS", which is very often criticized, mainly due to the lack of compatibility with the operating system and low speed of operation. However, this is a dialect extremely easy to learn and does not require the use of expanded Amiga configurations. For this reason, it is worth knowing it. We can also see how fast and simple is programming on your favorite computer.

This time I write about the language "Blitz Basic", for which you do not need to acquire many new skills. However, it is system compatible - it uses Workbench elements and the operating system. It allows you to directly create a typical user interface - screens, windows, various types of buttons or use the libraries. Of course, this is just one of the modes of operation. In the book, I focus primarily to show you how to write simple programs that display text or graphics in interaction with the operating system.

The second group of the commands disables the Amiga multitasking to get direct access to specialized chipset functions. I introduce this commands very slowly, explaining their functions and the implementation. I do not want to overwhelm the reader with a lot of information that may seem inconsistent from the point of view of a novice programmer. You have to be, dear Reader, aware of the different forms of action of your program, but we will be able to absorb all the information slowly, and without having to analyze complicated listings.

Many topics require continuation, which will take place in the next book on programming. I will present a different, more advanced point of view that requires more practice, even in the world of simple language, which is Basic, regardless of its variety.

Remember that programming is not only, as the definition says: the process of creating, designing and testing a computer program. In my opinion, this is primarily the art of improving one's own skills, which do not have to correspond to the trends prevailing in the environment of users of specific hardware. However they should "fit" to our goals, which can be ambitious, big or small, to finally write a program of everyday private use. So creativity in the first place.

Contrary to today's convictions of many people, I want to show that everyone can program, regardless of age and general interests. We can use the acquired experience in various operating systems, and the written programs will testify to our imagination and abilities of abstract thinking.

The book is based on the classic version of "Blitz Basic", which is the second generation working on Amiga computers. All we need is a 68000 processor, 1-2 MB of RAM and a hard drive. Theoretically, we can do without the latter, but I do not recommend such "fun", because it can quickly be discouraging. Thus, our programs will work on the usual Amiga 500, 600 or 1200. However, it should be remembered that this is not the only possibility to use the programming language, although I highly recommend starting with a simple computer configuration and uncomplicated programs. We can then find out for ourselves, how much the slogan associated with Amiga is true, such as below:

Remember when computing was fun?

Today, after 30 years from the release of Amiga, we can still enjoy its possibilities. For a person starting to programming it can be a whole big world that will not be exhausted too quickly. I wish my Readers to write their programs, games or demos, first of all for pleasure. Later, I invite you to show your achievements on the Internet or at one of many events related to retro computers. You do not have to constantly look for a something new, it's enough to look at what we have from a different point of view.

- September 2017 r.

BASICS

DOWNLOADING FROM THE INTERNET

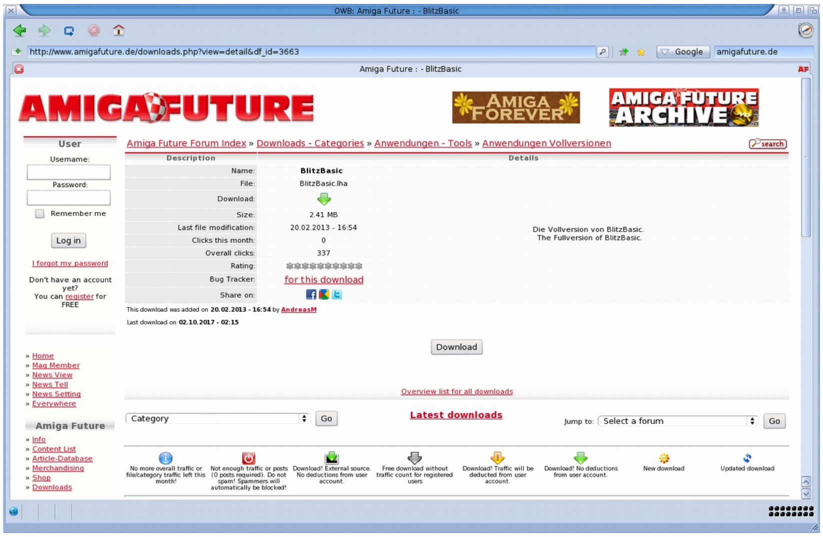
Blitz Basic can be obtained on one of many websites, including the following page:

`amigafuture.de`

Go to the "Downloads" section, "Anwendungen - Tools", then "Anwendungen Vollversionen" and search for "Blitz-Basic". You can also go directly to the page where you will find the archive. Here's the full address:

`http://www.amigafuture.de/downloads.php?view=detail&df_id=3663`

This is how the page looks when loaded into the OWB browser:


OWB: Amiga Future : - BlitzBasic
http://www.amigafuture.de/downloads.php?view=detail&df_id=3663
Google
amigafuture.de
Amiga Future : - BlitzBasic
AMIGA FUTURE
AMIGA FOREVER
AMIGA FUTURE ARCHIVE
search

User
Username:
Password:
Remember me
Log in
I forgot my password
Don't have an account yet?
You can register for FREE

» Home
» Mag Member
» News View
» News Tell
» News Setting
» Everywhere

Amiga Future
» Info
» Content List
» Article-Database
» Merchandising
» Shop
» Downloads

Amiga Future Forum Index » Downloads - Categories » Anwendungen - Tools » Anwendungen Vollversionen

Description	Details	
Name:	**BlitzBasic**	Die Vollversion von BlitzBasic. The Fullversion of BlitzBasic.
File:	BlitzBasic.lha	
Download:		
Size:	2.41 MB	
Last file modification:	20.02.2013 - 16:54	
Clicks this month:	0	
Overall clicks:	337	
Rating:		
Bug Tracker:	for this download	
Share on:		

This download was added on **20.02.2013 - 16:54** by **AndreasM**

Last download on **02.10.2017 - 02:15**

Download

Overview list for all downloads

Category Go

Latest downloads

Jump to: Select a forum Go

No more overall traffic or file/category traffic left this month!
Not enough traffic or posts (0 posts required). Do not spam! Spammers will automatically be blocked!
Download! External source. No deductions from user account.
Free download without traffic count for registered users
Download! Traffic will be deducted from user account.
Download! No deductions from user account.
New download
Updated download

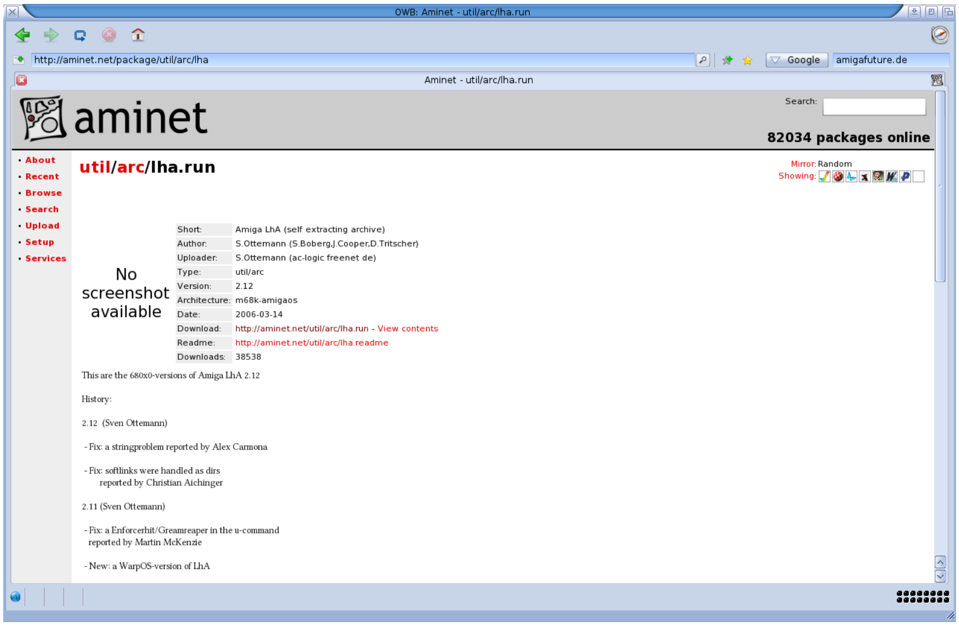
Now click the "Download" link and download the file to disk, to any directory. The archive should be unpacked using the LHA program, which can be downloaded from the website:

`aminet.net`

This time, we search for the phrase "lha.run" to find a file that can be unpacked automatically, without installing any more tools. The above name must be entered in the "Search" field or use the following address:

`http://aminet.net/package/util/arc/lha`

The page should look like this:



util/arc/lha.run

Short:	Amiga LhA (self extracting archive)
Author:	S.Ottemann (S.Boberg,J.Cooper,D.Tritscher)
Uploader:	S.Ottemann (ac-logic freenet de)
Type:	util/arc
Version:	2.12
Architecture:	m68k-amigaos
Date:	2006-03-14
Download:	http://aminet.net/util/arc/lha.run - View contents
Readme:	http://aminet.net/util/arc/lha.readme
Downloads:	38538

```
This are the 680x0-versions of Amiga LhA 2.12

History:

2.12  (Sven Ottemann)

 - Fix: a stringproblem reported by Alex Carmona

 - Fix: softlinks were handled as dirs
        reported by Christian Aichinger

2.11 (Sven Ottemann)

 - Fix: a Enforcerhit/Greamreaper in the u-command
   reported by Martin McKenzie

 - New: a WarpOS-version of LhA
```

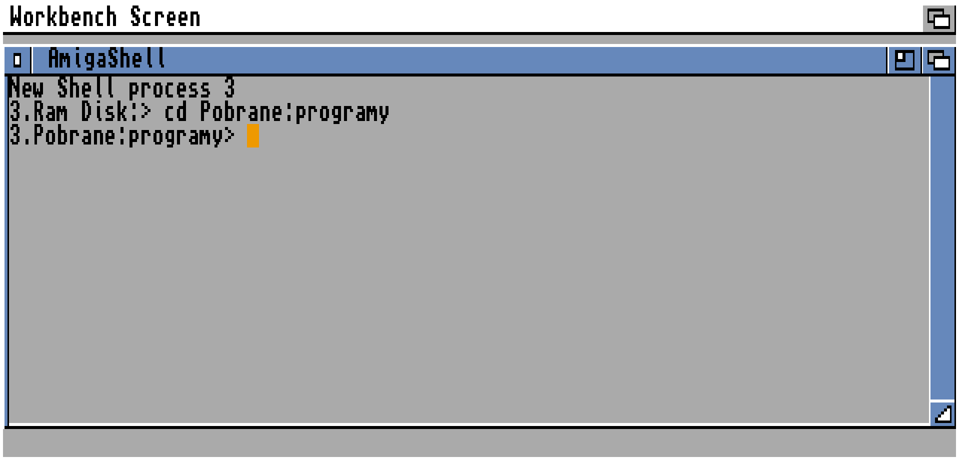
Click the name next to "Download:" in the box on the picture. The file will be downloaded to the disk. On Workbench, call the "Execute Command" option from the pull-down menu called "Workbench". Then type:

```
newshell
```

and press ENTER. The larger window signed "AmigaShell" will appear on the screen. Now you have to change the current directory to the one where you downloaded the files. To do this, type the CD command, then press the SPACEBAR and enter the full path. For example, if the files were saved in the "programs" directory on the "Downloaded" disk, the whole line should be in the following form:

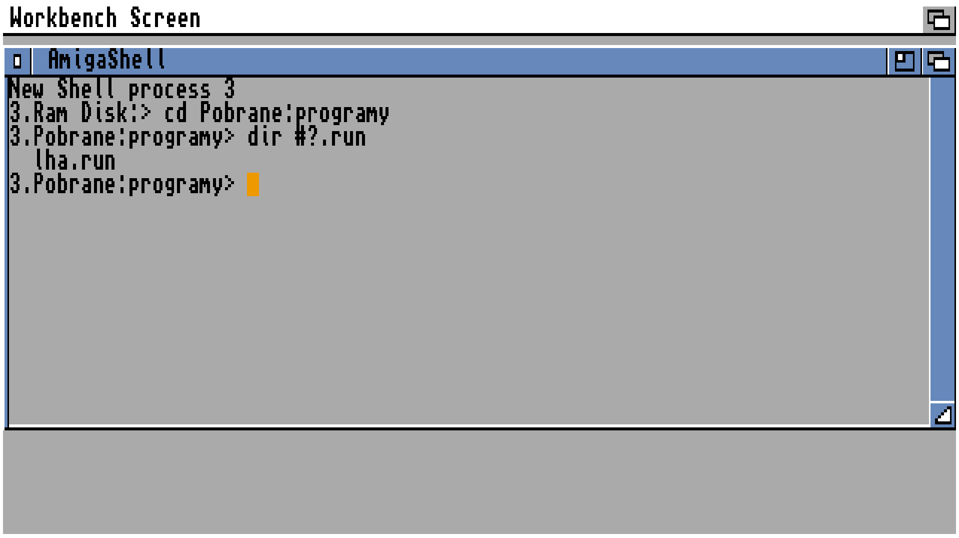
```
cd Downloaded:programs
```

After pressing the ENTER key, the text in front of the cursor should change as shown:



```
New Shell process 3
3.Ram Disk:> cd Pobrane:programy
3.Pobrane:programy>
```

Of course, if your directories have different names, you'll see a different string. However, it must always point to the directory where you saved the downloaded files. To be sure, you can display the contents of the directory using the DIR command. Just enter its name and the symbol "#?.run". Thanks to this, in the window you will see only files ending with the extension ".run". After pressing ENTER the information should look like this:



```
New Shell process 3
3.Ram Disk:> cd Pobrane:programy
3.Pobrane:programy> dir #?.run
  lha.run
3.Pobrane:programy>
```

When we are sure that we have access to the files, we can proceed with further work. In the "Shell" window, enter:

```
lha.run
```

and press ENTER. After a while new messages will appear and most of the lines should start with the "LHA" name. The last line should contain the following message:

```
Operation successfull
```

If the situation is different, it means that the file has not been saved correctly and has lost the so-called AmigaDOS attributes. Thanks to them, the file has specific features, among others the system determines whether a given file can be unpacked by entering its name - as in this case. To fix the problem you need to enter one more line, this time with the PROTECT command. Just type:

```
protect lha.run +rwed
```

Now the file will get the appropriate attributes and you will be able to use it. If everything is well, copy one of the files to the "C" system directory . To do this, enter another line:

```
copy lha_68k C:lha
```

After pressing the ENTER key, the cursor should go to the next line without displaying additional information. It means everything is fine. In this way, you have installed the LHA program on your system drive. With its use, you will unpack the archive with the Blitz Basic language.

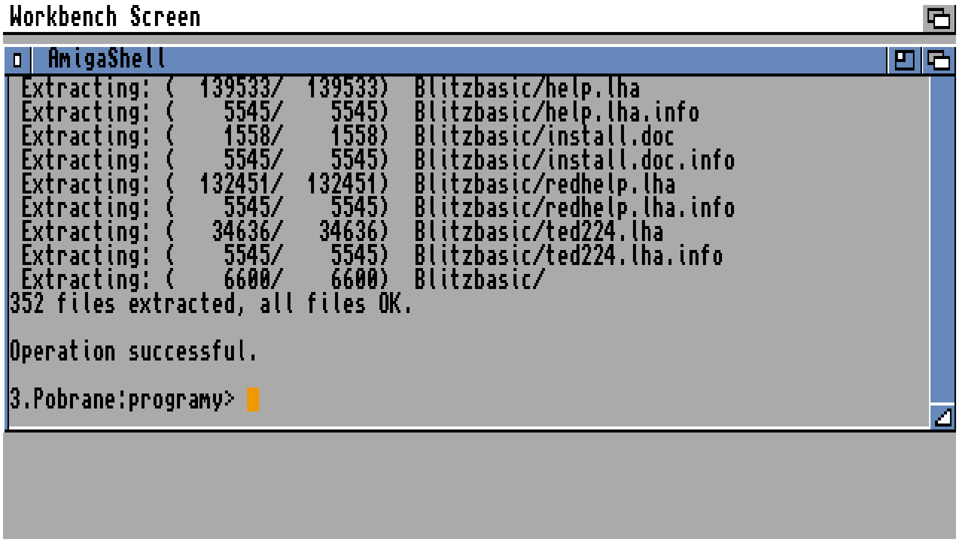
For this to happen, enter the line that will unpack the "BlitzBasic.lha" file to the hard disk or to the computer's memory, that is "Ram Disk". The easiest way is to use the same place where the files have been saved. Therefore, in the next step enter the line:

```
lha x BlitzBasic.lha
```

and confirm - as before - with the ENTER key. The window will scroll through many more file names, but in the last line will be again the message:

```
Operation successful
```

The situation should be similar to the following:



```
 Extracting: (  139533/  139533)  Blitzbasic/help.lha
 Extracting: (    5545/    5545)  Blitzbasic/help.lha.info
 Extracting: (    1558/    1558)  Blitzbasic/install.doc
 Extracting: (    5545/    5545)  Blitzbasic/install.doc.info
 Extracting: (  132451/  132451)  Blitzbasic/redhelp.lha
 Extracting: (    5545/    5545)  Blitzbasic/redhelp.lha.info
 Extracting: (   34636/   34636)  Blitzbasic/ted224.lha
 Extracting: (    5545/    5545)  Blitzbasic/ted224.lha.info
 Extracting: (    6600/    6600)  Blitzbasic/
352 files extracted, all files OK.

Operation successful.

3.Pobrane:programy>
```

All basic files are already unpacked to install Blitz Basic on your hard drive. How to do it you will learn from reading the next section.

INSTALLATION ON HARD DISK

The process of installing the Blitz Basic is quite complicated, because there is no program that could do it automatically for us. All operations can be done using one of the file managers like "Directory Opus" or "File Master", but we will do it without the installation of additional software. This is slightly less convenient procedure, but you can use it on any computer, even if you have just installed the operating system without any extensions.

At first, we need to create a new directory in which we will save all the files necessary for work. You can use the directory on any disk, as well as its name does not matter. We will use the short name "Blitz" to facilitate further operations. The data will be saved on the system disk in the "Tools" directory, which belongs to the standard items after installing the operating system.

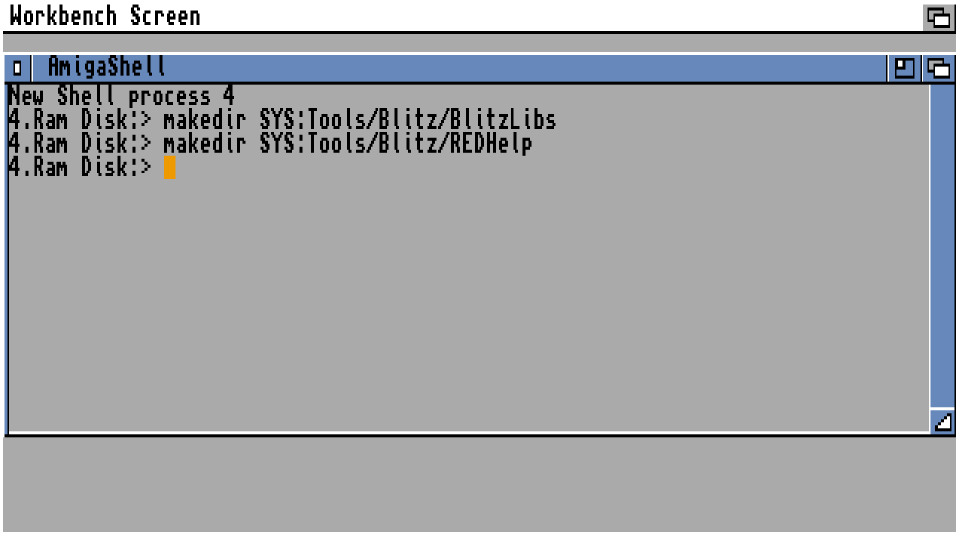
We will create new directory using the MAKEDIR command. Enter the following line:

```
makedir SYS:Tools/Blitz
```

and press ENTER. The next step will be to create two more directories inside "Blitz". To achieve this, enter the next two lines:

```
makedir SYS:Tools/Blitz/BlitzLibs
makedir SYS:Tools/Blitz/REDHelp
```

Please note that after executing each line in the "Shell" window, no additional message should appear. In this case, it would mean the error. So we know that everything went well, as in the picture:



```
New Shell process 4
4.Ram Disk:> makedir SYS:Tools/Blitz/BlitzLibs
4.Ram Disk:> makedir SYS:Tools/Blitz/REDHelp
4.Ram Disk:>
```

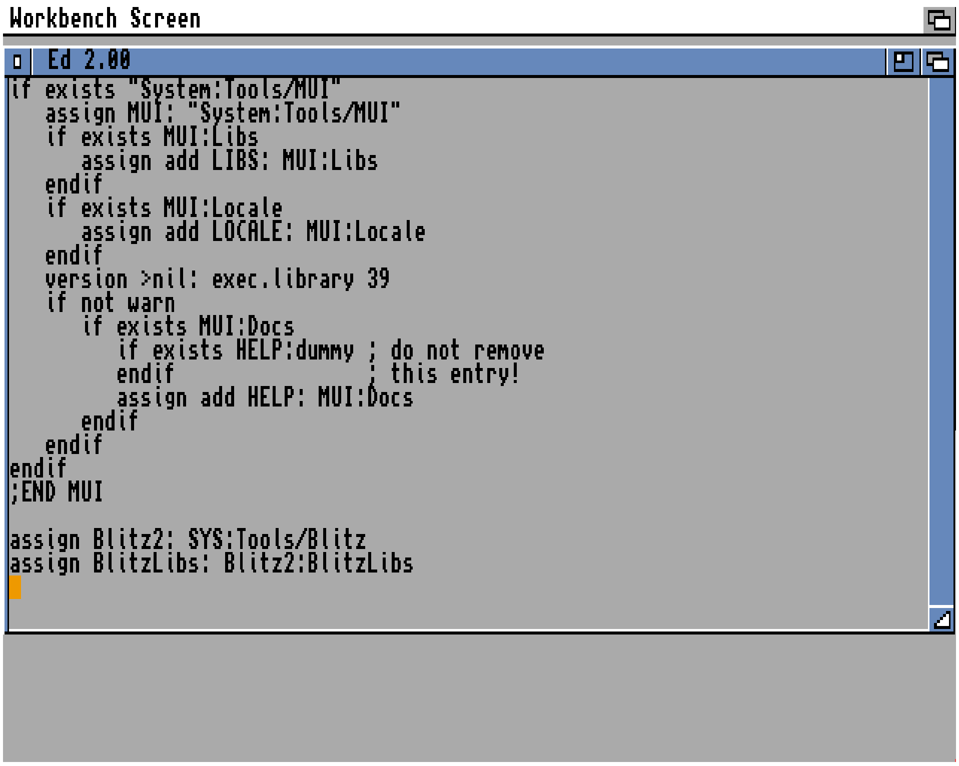
Now two logical devices, named "Blitz2:" and "BlitzLibs:" should be assigned to created directories. To do this, you have to edit the "user-startup" file, which is located in the "S" system directory. Therefore, enter the next line:

```
ed S:user-startup
```

and press ENTER. A large window will appear on the screen with the contents of the file. Move the cursor to the end and add two lines:

```
assign Blitz2: SYS:Tools/Blitz
assign BlitzLibs: Blitz2:Blitzlibs
```

You must do it very carefully so as not to mistake any sign, otherwise our "assigns" will not work. Select the "Save" option from the "Project" pull-down menu to save the file. The "Ed" program window should look like this:

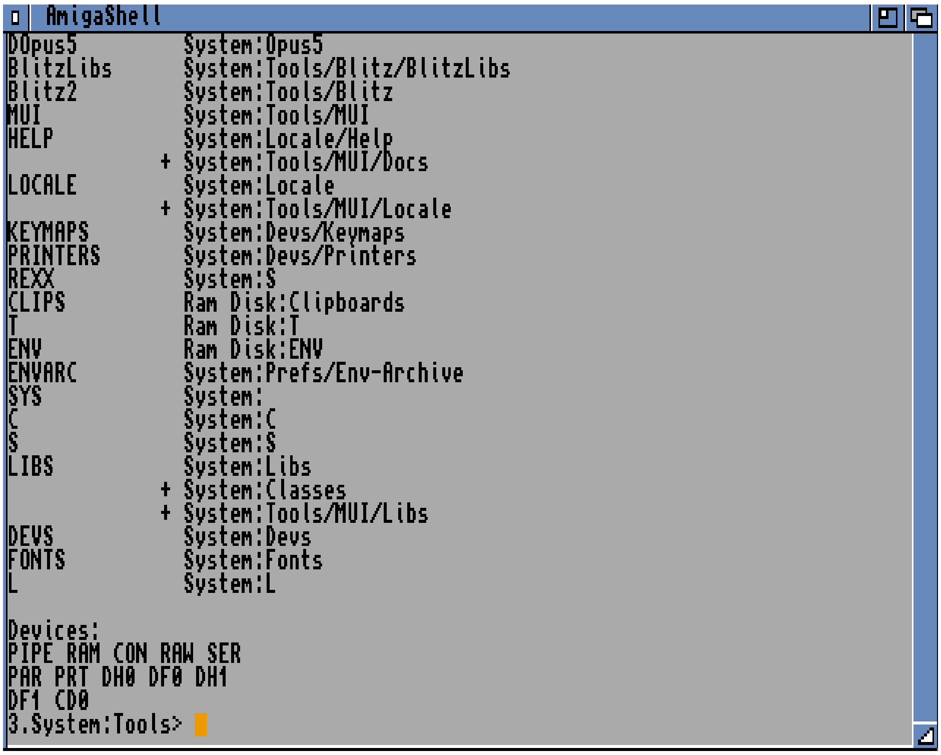


```
if exists "System:Tools/MUI"
   assign MUI: "System:Tools/MUI"
   if exists MUI:Libs
      assign add LIBS: MUI:Libs
   endif
   if exists MUI:Locale
      assign add LOCALE: MUI:Locale
   endif
   version >nil: exec.library 39
   if not warn
      if exists MUI:Docs
         if exists HELP:dummy ; do not remove
         endif                ; this entry!
         assign add HELP: MUI:Docs
      endif
   endif
endif
;END MUI

assign Blitz2: SYS:Tools/Blitz
assign BlitzLibs: Blitz2:BlitzLibs
```

Close the window, reset the Amiga and re-load the Workbench. If there are no error messages on the screen, it means that everything is ok. To be sure, you can check if the new logical devices are active. Open the "Shell" window again, enter:

```
assign
```

and press ENTER. This will display a list of available devices. Expand the window so that you can see the entire list and pay attention to the items beginning with the word "Blitz". For example:


```
AmigaShell
DOpus5         System:Opus5
BlitzLibs      System:Tools/Blitz/BlitzLibs
Blitz2         System:Tools/Blitz
MUI            System:Tools/MUI
HELP           System:Locale/Help
             + System:Tools/MUI/Docs
LOCALE         System:Locale
             + System:Tools/MUI/Locale
KEYMAPS        System:Devs/Keymaps
PRINTERS       System:Devs/Printers
REXX           System:S
CLIPS          Ram Disk:Clipboards
T              Ram Disk:T
ENV            Ram Disk:ENV
ENVARC         System:Prefs/Env-Archive
SYS            System:
C              System:C
S              System:S
LIBS           System:Libs
             + System:Classes
             + System:Tools/MUI/Libs
DEVS           System:Devs
FONTS          System:Fonts
L              System:L

Devices:
PIPE RAM CON RAW SER
PAR PRT DH0 DF0 DH1
DF1 CD0
3.System:Tools>
```

If both items are visible, you can proceed with the installation. Change the current directory to the same as before - on our drive it is the "Downloaded:programs" path. We have unpacked the LHA archive before, so we can use the files right away.

Use the following line to change the directory to "Blitzbasic" saved inside:

```
cd Blitzbasic
```

Now, if you display content using the DIR command, you will see more files with the ".lha" extension. Their contents should be unpacked to two specific directories. You have to do it the same way as before, that is, by entering the line according to the scheme:

```
lha x <FILE> <DIRECTORY>
```

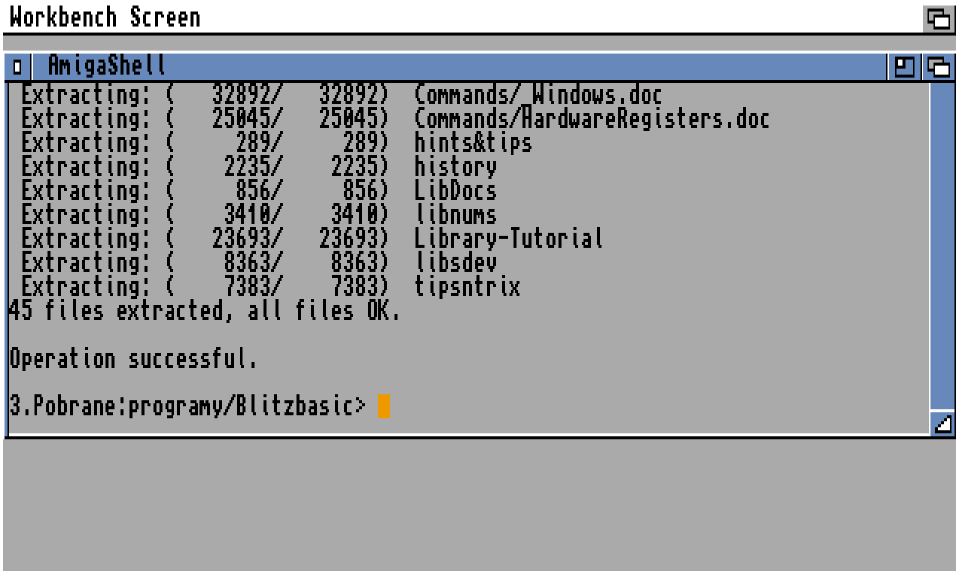
Of course, instead of "<FILE>" and "<DIRECTORY>", you must specify the names of LHA files and the paths. Here is their list:

<PLIK>	<KATALOG>
acidlibs.lha	Blitz2:
blitz210.lha	Blitz2:
blitzlibs.lha	Blitz2:Blitzlibs/
defdebug.lha	Blitz2:
deflibs.lha	Blitz2:
help.lha	Blitz2:
redhelp.lha	Blitz2:REDHelp/
ted224.lha	Blitz2:

You must enter a separate line for each file. In other words, to unpack the first file, type:

```
lha x acidlibs.lha Blitz2:
```

Each time new information will appear in the window, just like before. For example, like this:

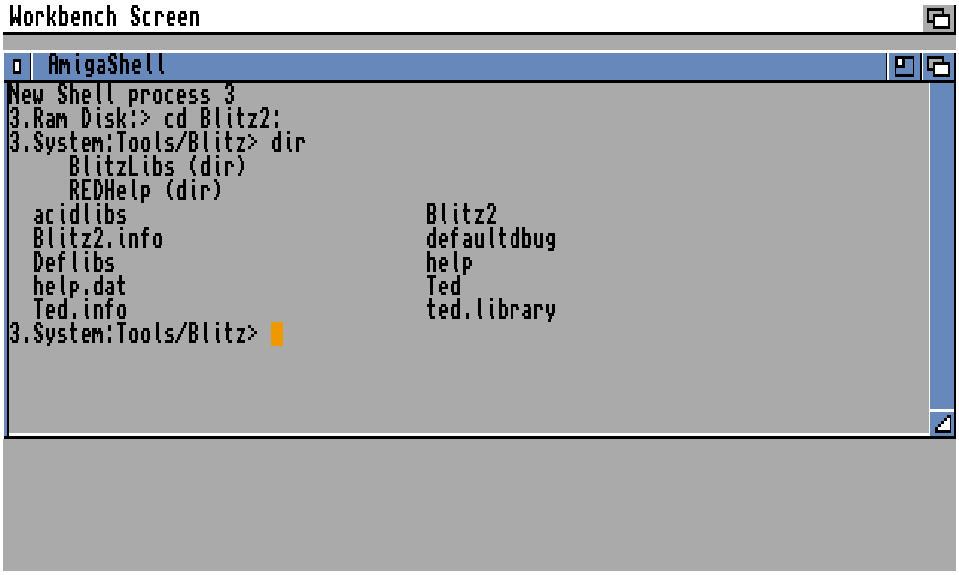


```
 Extracting: (   32892/   32892)  Commands/ Windows.doc
 Extracting: (   25045/   25045)  Commands/HardwareRegisters.doc
 Extracting: (     289/     289)  hints&tips
 Extracting: (    2235/    2235)  history
 Extracting: (     856/     856)  LibDocs
 Extracting: (    3410/    3410)  libnums
 Extracting: (   23693/   23693)  Library-Tutorial
 Extracting: (    8363/    8363)  libsdev
 Extracting: (    7383/    7383)  tipsntrix
45 files extracted, all files OK.

Operation successful.

3.Pobrane:programy/Blitzbasic>
```

As a result, in the "Blitz2:" logical device you should get the appropriate directory structure, and the files saved inside. To check this, change the current directory:

```
cd Blitz2:
```

and display its contents using the DIR command. You should see the following items:



```
New Shell process 3
3.Ram Disk:> cd Blitz2:
3.System:Tools/Blitz> dir
     BlitzLibs (dir)
     REDHelp (dir)
  acidlibs                          Blitz2
  Blitz2.info                       defaultdbug
  Deflibs                           help
  help.dat                          Ted
  Ted.info                          ted.library
3.System:Tools/Blitz>
```

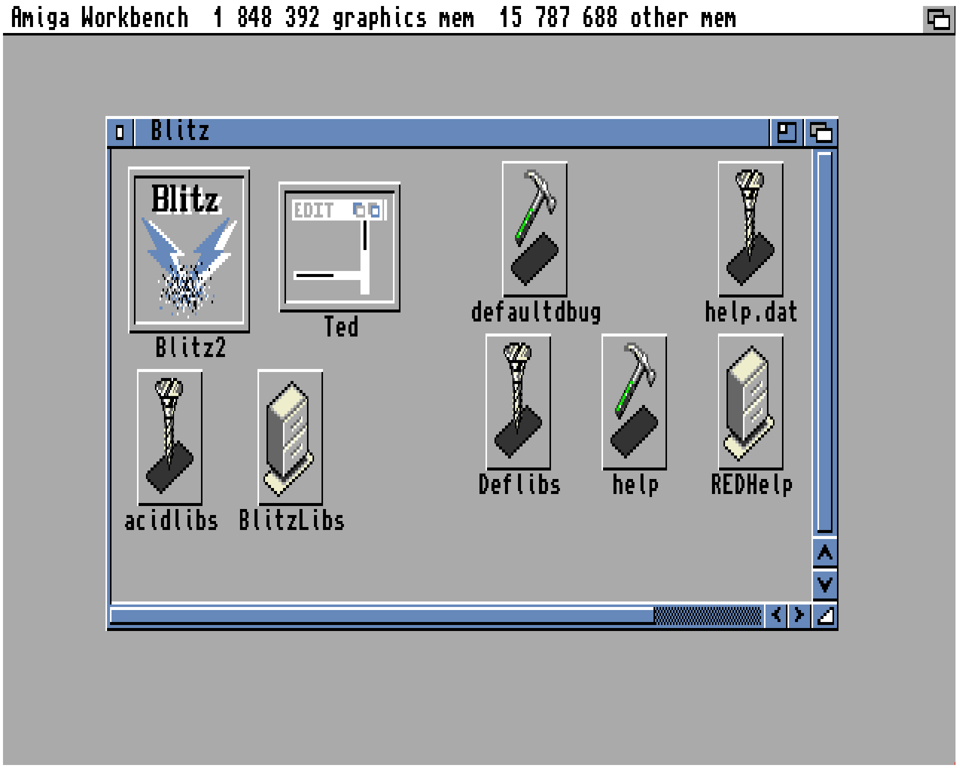
Finally, copy the system library "ted.library" to the "LIBS:" device. Enter another line:

`copy ted.library LIBS:`

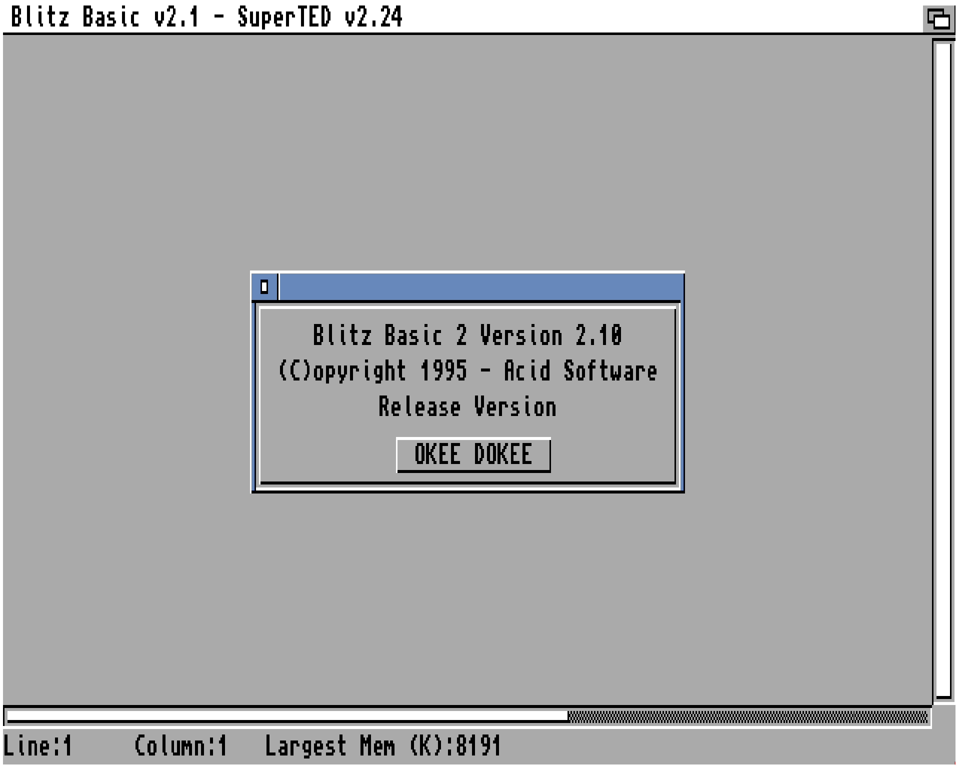
and press ENTER. The cursor should jump to the next line without additional messages.

Reset the Amiga and reload the Workbench. Read the contents of the system disk, then the "Tools" and the next "Blitz" directory that we have created. It does not have its own icon, so you will not see it right away. In the "Tools" directory window, call the pull-down menu "Window", select "Show" and then the "All files" option.

Scroll content of the window and find the "Blitz" icon. Make a simple double-click on it to read the content. You will see a new window and several icons inside. It should look like the following illustration:



Hover the pointer over the "Blitz2" icon and press the left mouse button twice. In this way, you will launch an editor in which you will be able to write Blitz Basic programs. Look again at the picture:


Blitz Basic v2.1 - SuperTED v2.24

Blitz Basic 2 Version 2.10
(C)opyright 1995 - Acid Software
Release Version

Line:1 Column:1 Largest Mem (K):8191

Select the "OKEE DOKEE" button. The editor is now ready to work. However before you start writing your own program, you should familiarize yourself with the basic functions of the program visible on the screen. You will read about it in the chapter entitled "Editor".

BOOTING FROM THE FLOPPY DISK

Working only with a floppy disk drive is not convenient. If your Amiga has the option of connecting a hard drive, do not hesitate to buy even the smallest drive. The speed of reading and saving files and the convenience of using large size files can not be overestimated.

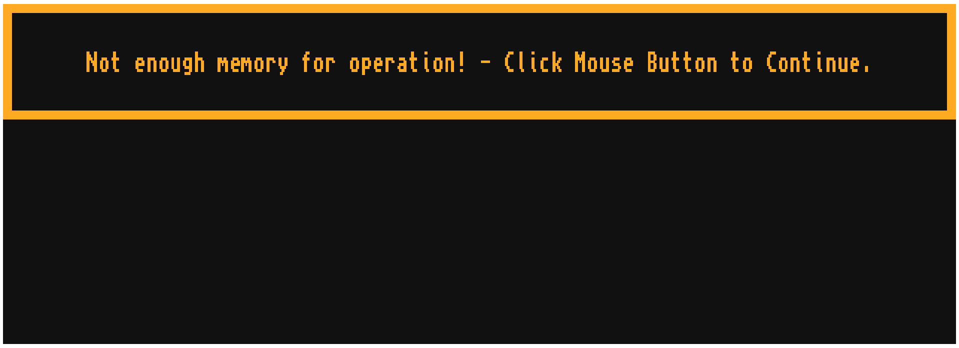
However, if for various reasons you have to limit yourself to floppy disks, start by making a copy of the system diskette from which you start the Workbench. Delete all unneeded files, for example the contents of the following directories:

```
System
Rexx
Utilities
```

On the floppy you will get the area for storing only essential Blitz Basic files. If your Amiga has at least approx. 2 MB of free memory, you can copy the necessary files to the "Ram Disk" and use it as an alternative to loading the entire hard disk.

All activities must be performed in a similar way to the description from the previous section. You can also save the files on a CD, then add the lines with the ASSIGN command to the "startup-sequence" file, and activate the optical drive driver before.

However in practice this work will not make sense, because in problematic situations you will have to repeat most of the procedure. In addition, you can see a message indicating that there is not enough free memory to work:

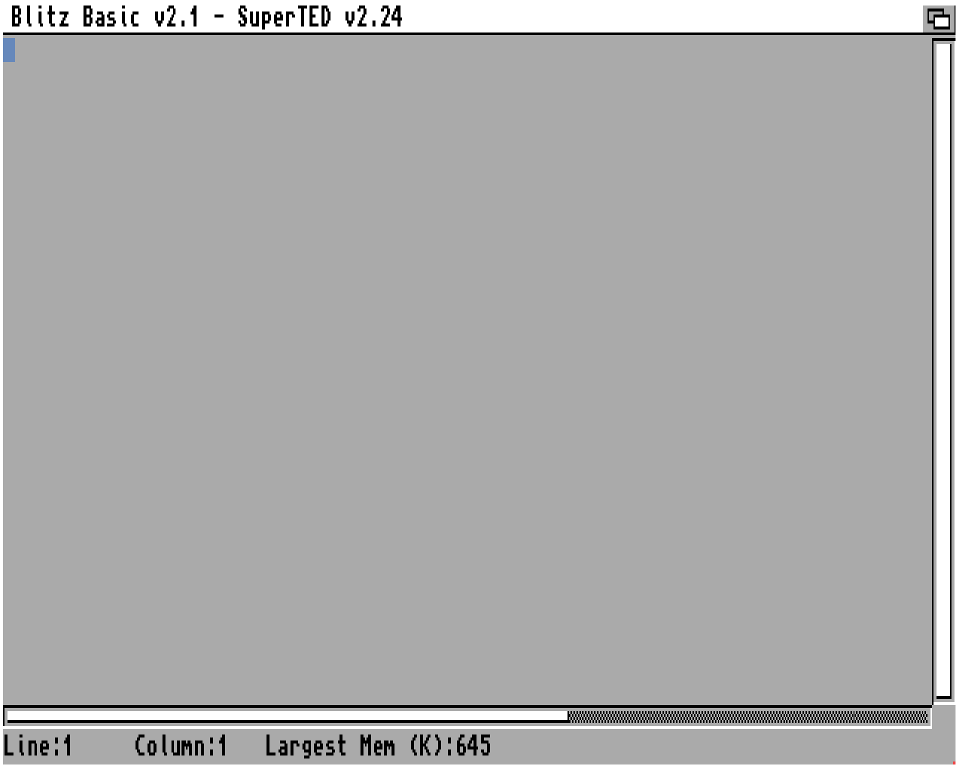


```
Not enough memory for operation! - Click Mouse Button to Continue.
```

Therefore, as with other computers, if you want to deal with the more serious use of Amiga you need to equip it with a hard disk controller and the appropriate drive. In this situation, even a memory of 1-2 MB will not limit the ability to load the editor, but you will have to start the computer without executing the "startup-sequence" and manually perform the "assignment" to the logical device called "ENV". The easiest way to do this is by using the line:

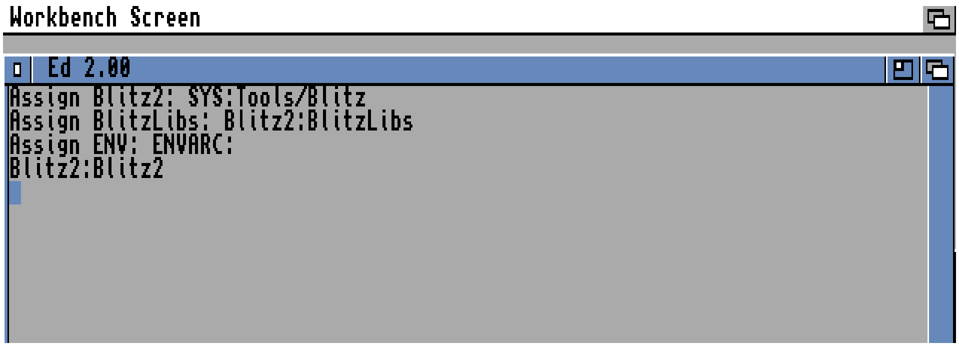
```
assign ENV: ENVARC:
```

In addition, you must create logical devices with the names "Blitz2" and "BlitzLibs", just like before, but outside the "user-startup" file. In this way, you will save memory and make the area around 600 kilobytes remain accessible. As in our illustration:


Blitz Basic v2.1 - SuperTED v2.24

Line:1 Column:1 Largest Mem (K):645

For simplicity, see the modified "startup-sequence" file, which loads only those components that are absolutely necessary for operation:


Workbench Screen

Ed 2.00

```
Assign Blitz2: SYS:Tools/Blitz
Assign BlitzLibs: Blitz2:BlitzLibs
Assign ENV: ENVARC:
Blitz2:Blitz2
```

Your file should look similar, because these lines create only the mentioned logic devices, and then launch the Blitz Basic editor. How to operate it we will say in the next chapter.

EDITOR

TYPING THE PROGRAM

The editor you see on the screen is called "SuperTED" and is a developed version of the "Ted" program. This is the main element where we write the program and perform other activities related to testing and modification of commands. Remember that you do not have to use all the possibilities right away, the best way is to get to know the functions you need at the moment.

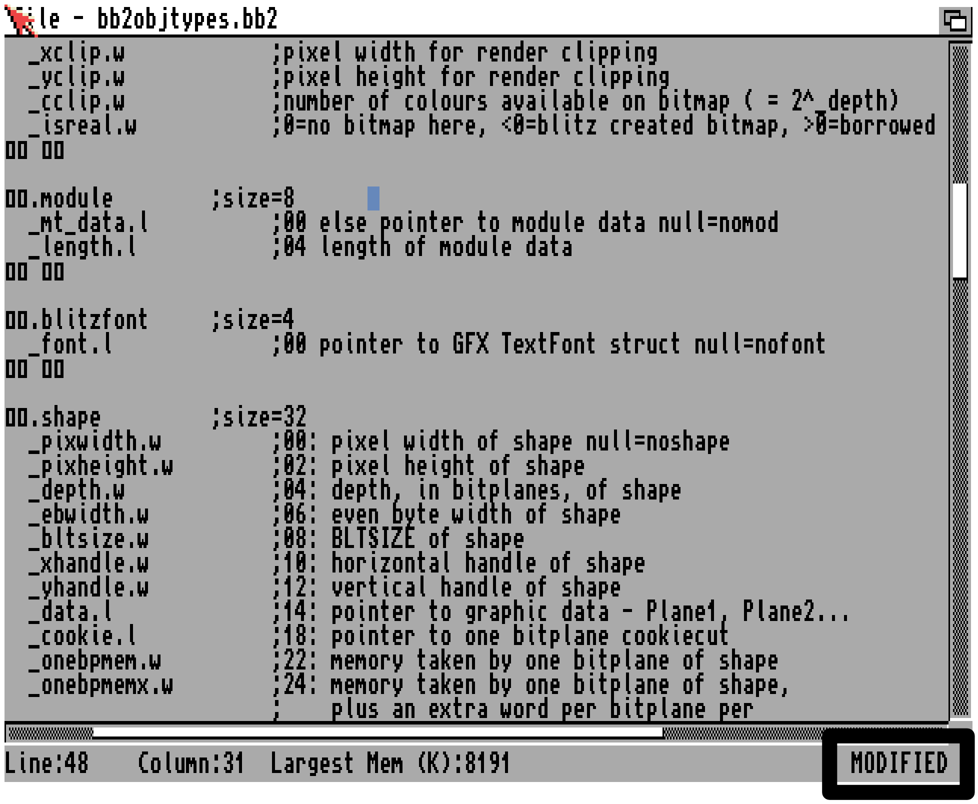
"Ted" is an ordinary text editor, so you can use it to save any ASCII files. The workspace is not fundamentally different from other similar programs. A standard window, scroll bars and other typical elements are visible. They all work similarly to the windows of other programs run on the Workbench screen.

Individual options related to the Blitz Basic language can be found in the pull-down menu of the program. Before you learn how to use them, let's add that it is worth knowing keyboard shortcuts, which you can use to call functions more quickly or easily.

Please note that the program is similar to the "Ed" system editor, but it is more extensive. The bottom part of the screen contains information about the cursor position - "Line" and "Column". They mark the line and column in which the cursor is currently located.

Next to it is a number indicating the maximum amount of memory you can use. It is a value expressed in kilobytes, so the entry "8191" means approx. 8 megabytes. On some illustrations you can notice a much larger number, because everything depends on the specific amount of computer memory. You can not use 100% memory, because the program must fit in a continuous area.

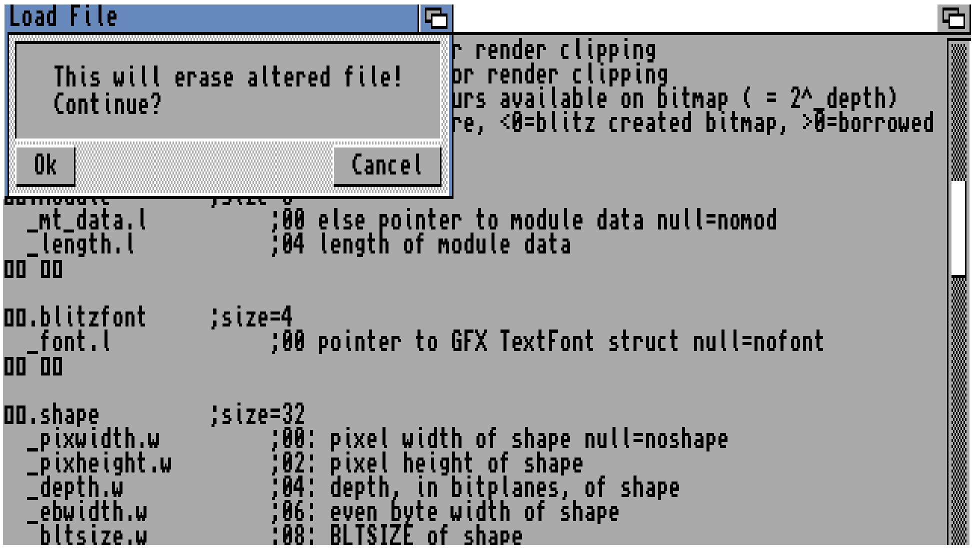
Please note that sometimes the word "MODIFIED" appears in the lower right corner:



```
  _xclip.w           ;pixel width for render clipping
  _yclip.w           ;pixel height for render clipping
  _cclip.w           ;number of colours available on bitmap ( = 2^_depth)
  _isreal.w          ;0=no bitmap here, <0=blitz created bitmap, >0=borrowed
□□ □□

□□.module       ;size=8
  _mt_data.l         ;00 else pointer to module data null=nomod
  _length.l          ;04 length of module data
□□ □□

□□.blitzfont    ;size=4
  _font.l            ;00 pointer to GFX TextFont struct null=nofont
□□ □□

□□.shape        ;size=32
  _pixwidth.w        ;00: pixel width of shape null=noshape
  _pixheight.w       ;02: pixel height of shape
  _depth.w           ;04: depth, in bitplanes, of shape
  _ebwidth.w         ;06: even byte width of shape
  _bltsize.w         ;08: BLTSIZE of shape
  _xhandle.w         ;10: horizontal handle of shape
  _yhandle.w         ;12: vertical handle of shape
  _data.l            ;14: pointer to graphic data - Plane1, Plane2...
  _cookie.l          ;18: pointer to one bitplane cookiecut
  _onebpmem.w        ;22: memory taken by one bitplane of shape
  _onebpmemx.w       ;24: memory taken by one bitplane of shape,
                     ;    plus an extra word per bitplane per
Line:48    Column:31   Largest Mem (K):8191                     MODIFIED
```

If you can see this message, it means that the file has been changed in relation to the version saved on the disk. If you save it anywhere, the message will disappear. Thanks to this, you always know whether the content of the program is the same as on the disk or has already been modified. Contrary to appearances, this is very important message, and it is worth remembering.

If you try to load another file or close the editor without saving the program, in the upper left corner you will see a small window like this:



```
This will erase altered file!
Continue?
```

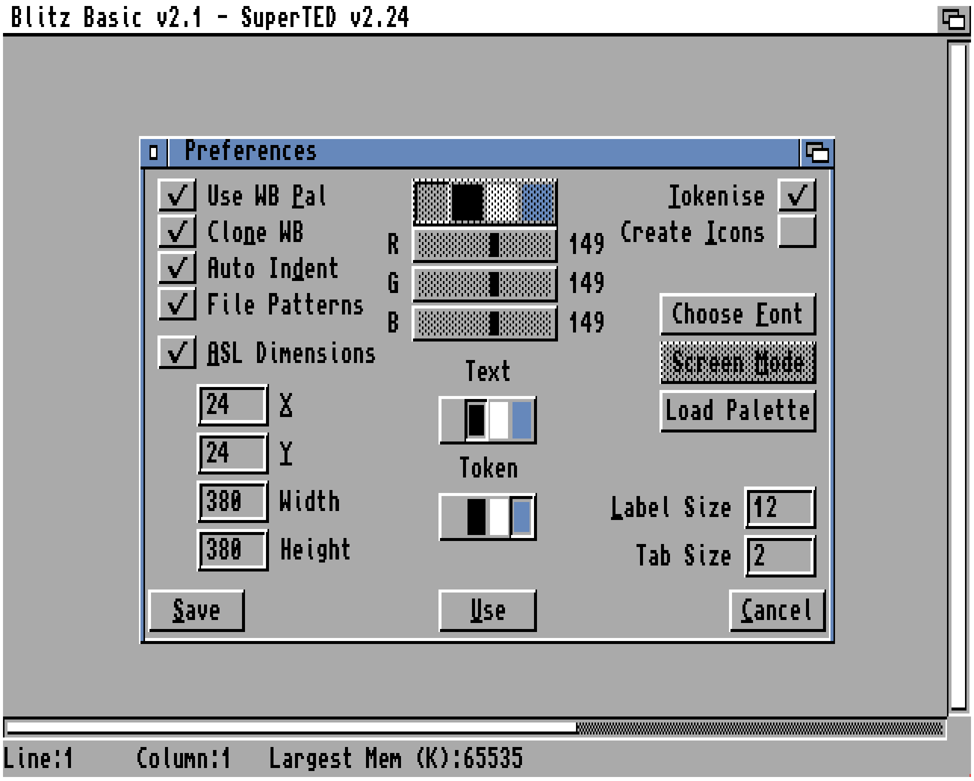
```
r render clipping
or render clipping
urs available on bitmap ( = 2^_depth)
re, <0=blitz created bitmap, >0=borrowed

  _mt_data.l          ;00 else pointer to module data null=nomod
  _length.l           ;04 length of module data
00 00

00.blitzfont     ;size=4
  _font.l             ;00 pointer to GFX TextFont struct null=nofont
00 00

00.shape         ;size=32
  _pixwidth.w         ;00: pixel width of shape null=noshape
  _pixheight.w        ;02: pixel height of shape
  _depth.w            ;04: depth, in bitplanes, of shape
  _ebwidth.w          ;06: even byte width of shape
  _bltsize.w          ;08: BLTSIZE of shape
```

This time, the editor informs you that you will lose your program. If it is not valid, use the "Ok" button and perform further operations, for example, load a different file. However, if you want to keep the contents of the editor, select the "Cancel" button and save the file to disk using the "Save As..." option. You will find it in the pull-down menu, which we will now discuss in more detail.

PULL-DOWN MENU

The most characteristic thing in the "Ted" editor is the pull-down menu, which contains a number of functions directly related to the operation of your program. The first one called "Project" contains entries that allow you to read and write files on the disk. These are typical options that can be found in most programs for Amiga.

A special feature is "Prefs...", thanks to which you can change the program settings. The configuration window fills almost the entire screen:



Blitz Basic v2.1 - SuperTED v2.24

Preferences

✓ Use WB Pal
✓ Clone WB
✓ Auto Indent
✓ File Patterns
✓ ASL Dimensions

24 X
24 Y
380 Width
380 Height

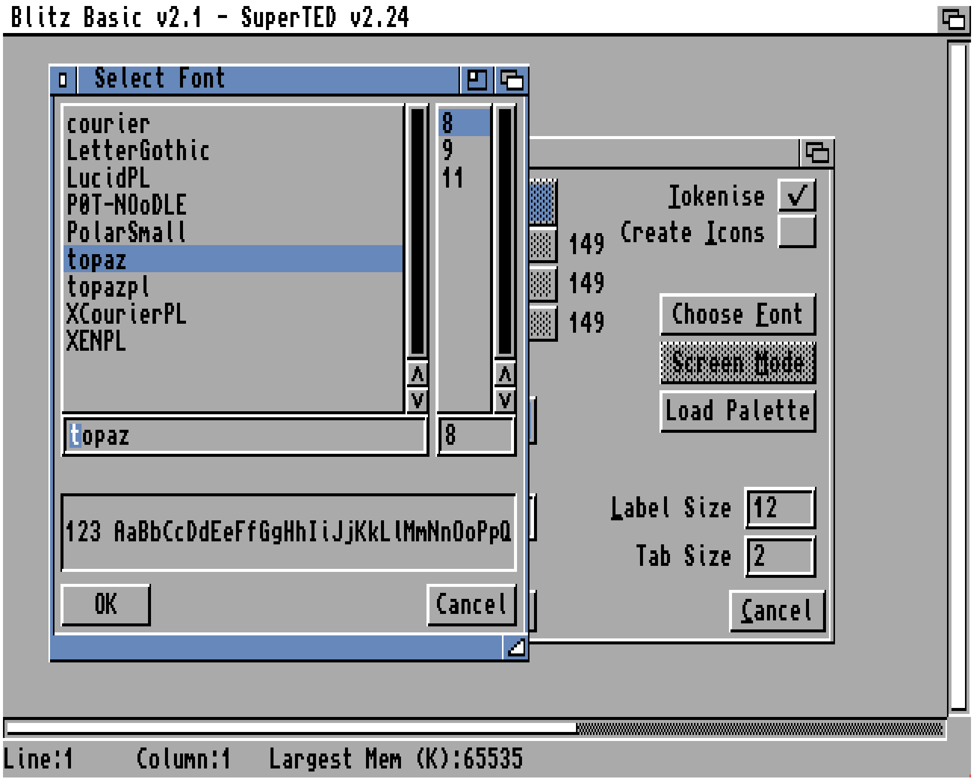
R 149
G 149
B 149

Tokenise ✓
Create Icons

Label Size 12
Tab Size 2

Line:1 Column:1 Largest Mem (K):65535

You do not have to pay attention to all the options, but it's worth to get to know the most important ones. Using the "Choose Font" button it is possible to choose a font that will be used for the entire editor. On the screen you will see a standard font selection window with the "Select Font" title, as in the following illustration:



Indicate the font you are interested in and confirm with the "OK" field. The editor screen will be updated and all its elements will use the new font. This will be taken into account for both the information line at the bottom and the content of the program.

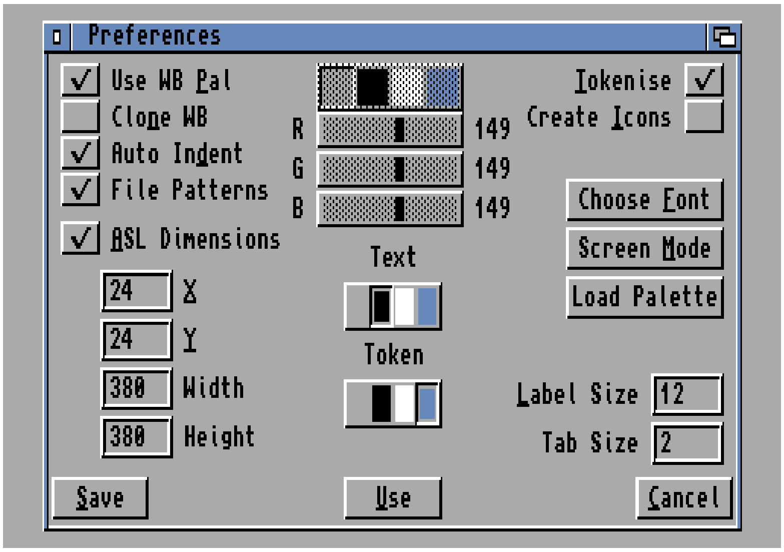
Please note that the configuration window will still use the standard "Topaz" fonts. Thanks to this you do not have to worry about whether you choose the right font, because even if you use very exotic characters, the contents of the "Preferences" window will always be readable.

Another important option is the field marked as "Create Icons". To activate them simply hover the pointer over the square and press the left mouse button. In the field a characteristic sign will be visible. Now, when saving files in the editor, the appropriate icons will be automatically added. Thanks to this, all you have to do is double-click them in the Workbench window to open both, the editor and the program's content saved in the file.

In the central part of the window, several larger colored boxes are visible, and "Text" message above them. Using them, you can change the color that your program will be displayed. Of course, this applies only to the content of the listing in the editor. Just click one of the boxes and then use the "Use" or "Save" button at the bottom. Let's add that the first one remembers the settings until the program is restarted, and the second saves the changes permanently.

A different color of text can be useful when writing a long program. Eyes quickly get used to new working conditions, but after a while you can feel eye strain. However, the four basic colors may not be enough for you, which is why the authors of "Ted" made it possible to run it on a separate screen.

Now you have to do two things. First, disable the option marked "Clone WB" on the left side of the window. When the field is inactive, the "Screen Mode" button on the right becomes active. See how it should look:

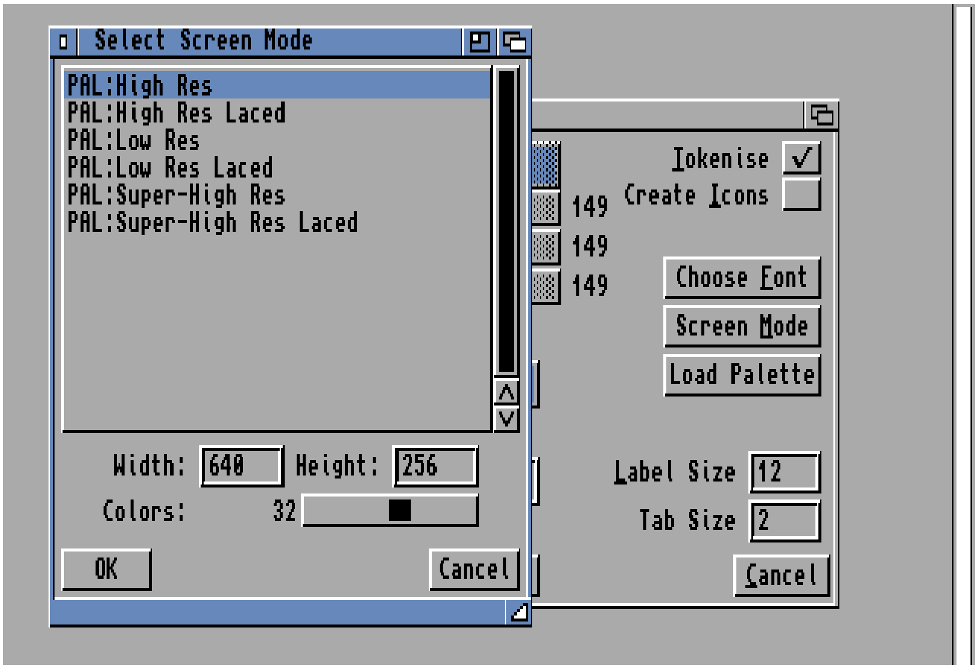


[✓] Use WB Pal
[ ] Clone WB
[✓] Auto Indent
[✓] File Patterns
[✓] ASL Dimensions
24 X
24 Y
380 Width
380 Height
R 149
G 149
B 149
Tokenise [✓]
Create Icons [ ]
Label Size 12
Tab Size 2

You must select the display mode that will be used on a separate screen assigned to the editor. Choose the mentioned button to display a list of available modes. Please note that, in addition to their names, at the bottom you can see the "Colors" slider with which you can change the number of available colors. Move it to the right to get at least 16 colors.

Remember that specific features depend on the chipset installed in your Amiga and the screen driver. The standard is "PAL", but you can also use the "Multiscan" or "Euro72" mode. You only need to turn them on the Workbench.

On the Amiga with AGA chipset you can set 32 colors. The window should look like this:


Select Screen Mode

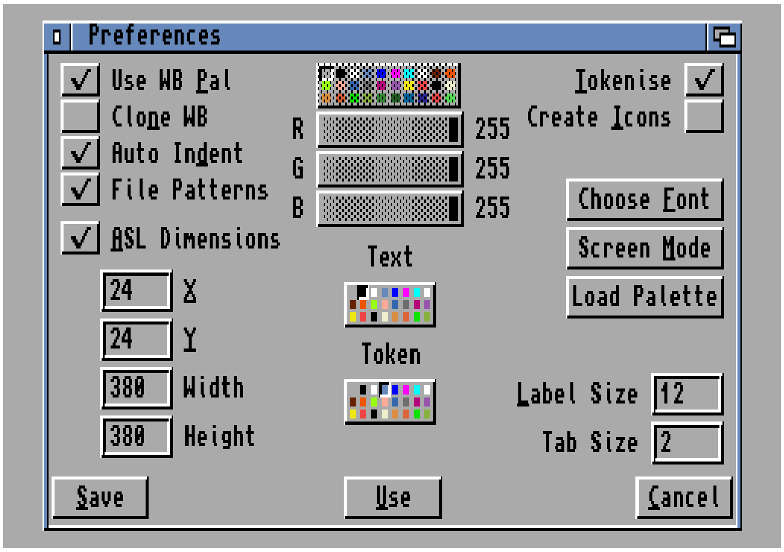
PAL:High Res
PAL:High Res Laced
PAL:Low Res
PAL:Low Res Laced
PAL:Super-High Res
PAL:Super-High Res Laced

Width: 640 Height: 256
Colors: 32

Select "OK" and then "Use" or "Save" in the main configuration window. The screen will be switched and "Ted" will appear on the separate screen with different parameters. Select the "Prefs..." option again from the pull-down menu to find out that is true.

We have not changed the resolution, so the text will look the same, but in the middle of the window you will see a larger number of colored boxes. Their function will remain the same, but the choice should be made more precisely.

You will not see all possible colors here, because the editor uses up to 24 colors. Look at the difference:


Preferences

✓ Use WB Pal
Clone WB
✓ Auto Indent
✓ File Patterns
✓ ASL Dimensions

24	X
24	Y
380	Width
380	Height

R 255
G 255
B 255

Text
Token

Tokenise ✓
Create Icons

Choose Font
Screen Mode
Load Palette

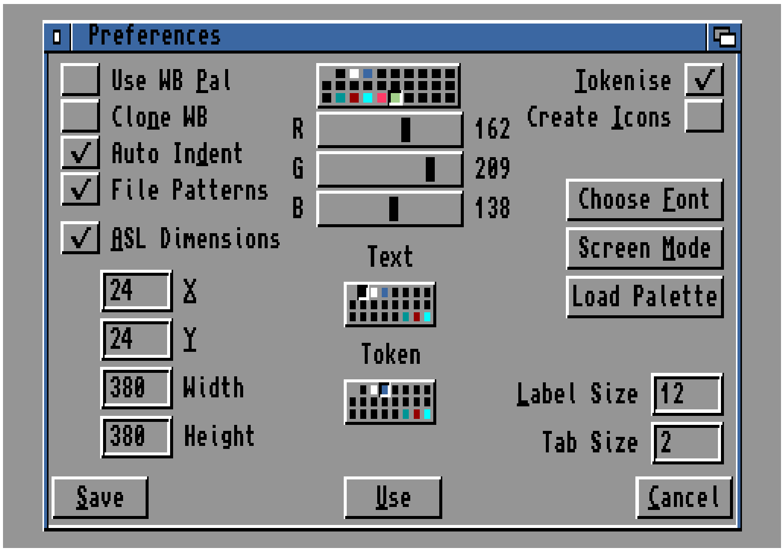
Label Size 12
Tab Size 2

Save Use Cancel

This choice is perfectly sufficient. Now you can freely decide on the color of your listing. Remember that the more colors on the screen, the more memory the editor takes. Therefore, use this option primarily when you have an Amiga equipped with an additional Fast memory, which at the time of starting the program is largely free.

If you want to get a color that you can not see, you can change the color palette. By default, the program accepts a palette the same as on the Workbench screen. It is enough, however, to turn off the button named "Use WB Pal" so that the situation changes. In this case, the screen color changes immediately after activating the field - without the need to confirm the operation.

If you want to restore your previous state, simply click the "Use WB Pal" box again. Along with this change, the sliders placed in the central part of the window will be active, described as "R", "G" and "B":



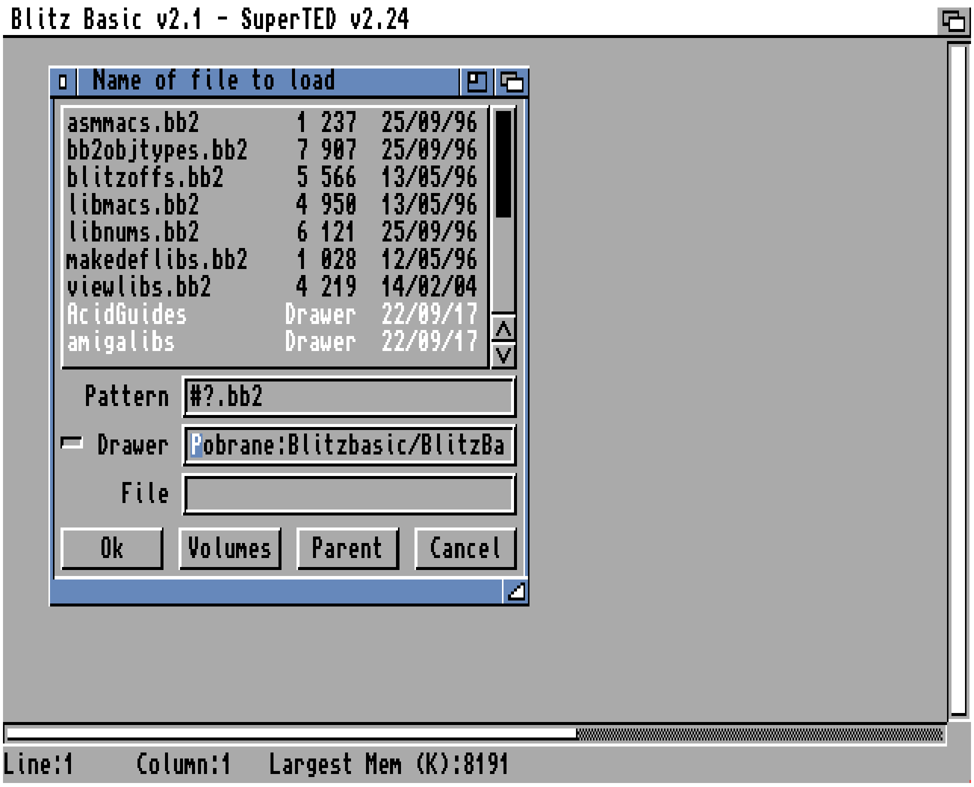
Use WB Pal [ ]
Clone WB [ ]
Auto Indent [✓]
File Patterns [✓]
ASL Dimensions [✓]
24 X
24 Y
380 Width
380 Height
R 162
G 209
B 138
Tokenise [✓]
Create Icons [ ]
Label Size 12
Tab Size 2

Now you can modify the colors. All you have to do is click on one of the boxes at the top - above the sliders - and change the state of one of the components: red (R), green (G) or blue (B). The editor screen can be adapted not only to the configuration of your Amiga, but also to the monitor. It's best to set colors that will not tiring to your eyes, but will not not make it difficult you the program analysis. Listing must always remain legible.

An important function is also the field marked as "File Patterns". If it is active, an additional "Pattern" field will appear in the file selection windows - after calling up the "Load", "Open" or "Save" option.

Thanks to it, you can use the so-called AmigaDOS filters, which are characters replacing names, for example "#?". This is useful when you have a lot of files saved in one directory on the disk and you want to display only a specific group. Not every window of choice gives such an option, in the case of the "Ted" editor you can decide about it.

So if in the directory you want to see only files with the extension ".bb2", which are characteristic for Blitz Basic, use the entry as in the following illustration:


Blitz Basic v2.1 - SuperTED v2.24
Name of file to load

```
asmmacs.bb2          1 237   25/09/96
bb2objtypes.bb2      7 907   25/09/96
blitzoffs.bb2        5 566   13/05/96
libmacs.bb2          4 950   13/05/96
libnums.bb2          6 121   25/09/96
makedeflibs.bb2      1 028   12/05/96
viewlibs.bb2         4 219   14/02/04
AcidGuides          Drawer   22/09/17
amigalibs           Drawer   22/09/17
```

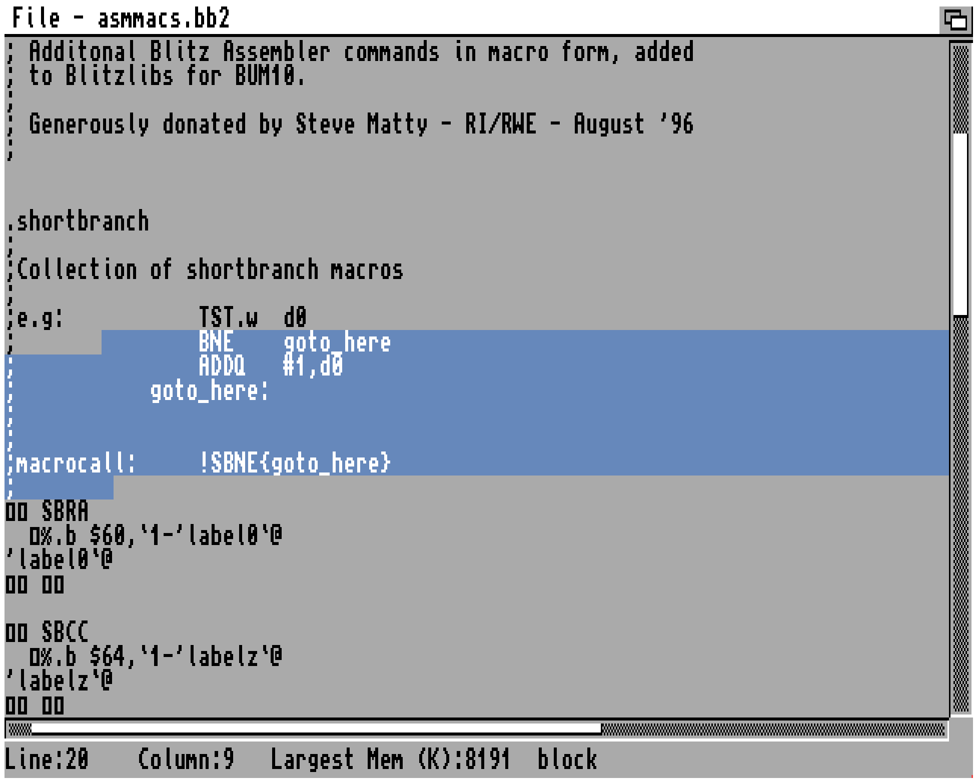
Pattern #?.bb2
Drawer Pobrane:Blitzbasic/BlitzBa
File

Ok
Volumes
Parent
Cancel
Line:1 Column:1 Largest Mem (K):8191

If the "File Pattens" is disabled, the selection window will be deprived of an additional field, and all saved files will be visible on the list.

OPERATION ON BLOCKS

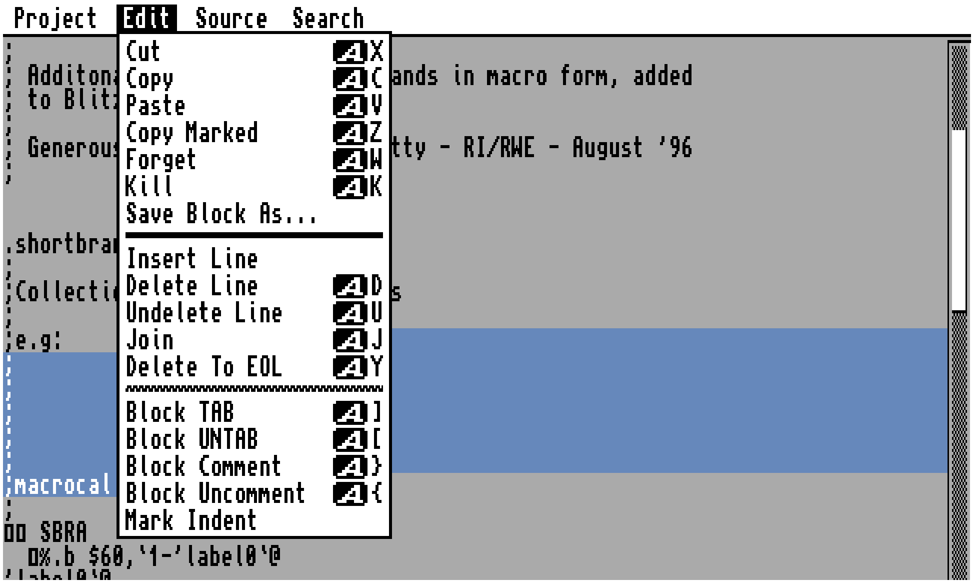
When editing a program, you often need to use blocks to move or copy larger fragments of text. We mark the blocks in the same way as in other editors, using the left mouse button. However, here you can do it using the keyboard itself. Just place the cursor at the beginning of the block, press the F1 key, then change the cursor position and use the F2 key. The block selected in this way looks like this:



```
File - asmmacs.bb2
; Additonal Blitz Assembler commands in macro form, added
; to Blitzlibs for BUM10.
;
; Generously donated by Steve Matty - RI/RWE - August '96
;


.shortbranch
;
;Collection of shortbranch macros
;
;e.g:           TST.w  d0
;               BNE    goto_here
;               ADDQ   #1,d0
;           goto_here:
;
;
;macrocall:     !SBNE{goto_here}
;
□□ SBRA
  □%.b $60,'1-'label0'@
'label0'@
□□ □□

□□ SBCC
  □%.b $64,'1-'labelz'@
'labelz'@
□□ □□
Line:20    Column:9   Largest Mem (K):8191   block
```

You may have the impression that the cursor disappears for a moment, but in reality it only merges with the end of the block, because it has the same color. Just use the cursor keys and move it down to see that it is set at the end of the selected block. The disadvantage of this solution is that when selecting - before you press F2 - you can not see the part that will be part of the block. Only later the part is highlighted. When selecting text with the mouse, this effect does not occur and all content is updated on a regular basis.

Of course, the marked block benefits from the Clipboard. You can perform many operations available in the pull-down menu called "Edit". Note that most of them are inactive before selecting the block. In addition, we can see here a lot more functions than usual in similar programs. The situation after block selection should look like this:



The first three items are typical functions that allow you to cut, copy and insert selected parts of the text in different places. We can also delete the selection using the "Forget" option or delete the block from the program content using "Kill".

If you want to save the selected fragment on the disk, use the "Save Block As..." function. It works similarly to saving the entire file, so a normal file selection window will appear on the screen. However, this time you only save a part of the program.

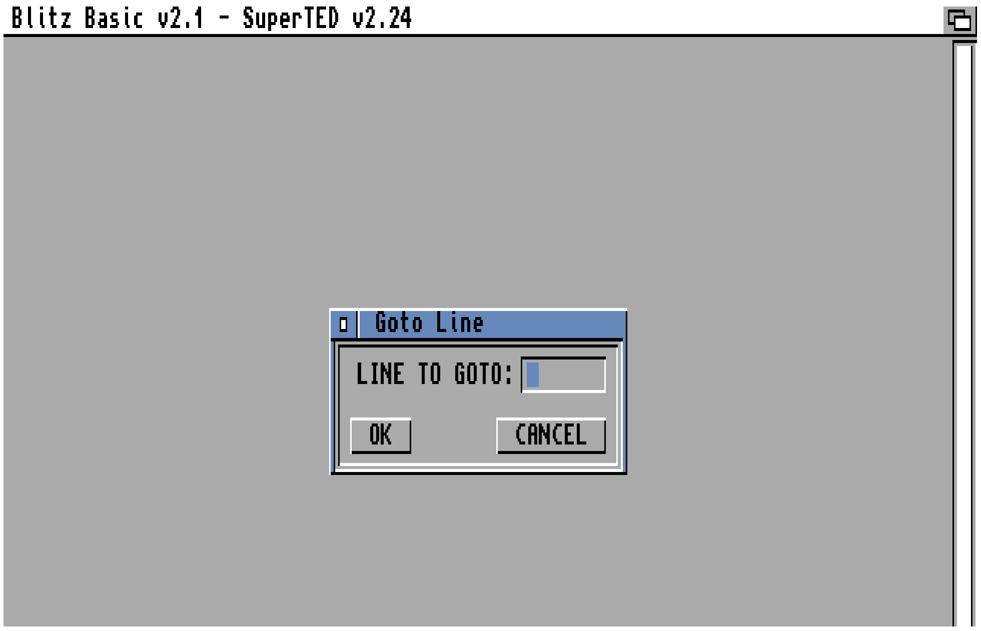
It is also possible to delete whole lines or insert empty ones. The two functions shown below are used for this purpose - "Delete Line" and "Insert Line". If you delete the line accidentally, you can recover it using the "Undelete Line" option. The function only works in relation to the last deleted line. If you call it multiple times, you will get multiple lines with the same content.

You can also connect the line with the next one, where the cursor is located. We do this using the "Join" function. Please note that it does not matter what is the horizontal position of the cursor, but only vertical. The lines are always combined taking into account their full content, so if at the beginning or the end there are SPACES, everything will be preserved in the new long line.

Thanks to the "Block TAB" and "Block UNTAB" functions, you can control the horizontal position of the lines included in the currently selected block. The first one shifts all the marked lines to the right, the second one moves in the opposite direction. By default, the size of this "margin" is 2 characters, but you can change it by entering a new value in the "Tab Size" field available in the configuration window.

You can move the cursor around the screen with the mouse or keyboard. There is also an pull-down menu called "Source", where you will find 3 important functions. First - "Top" - moves the cursor to the beginning of the loaded file. The second option - "Bottom" - works in reverse, namely it moves the cursor to the last line of the file.

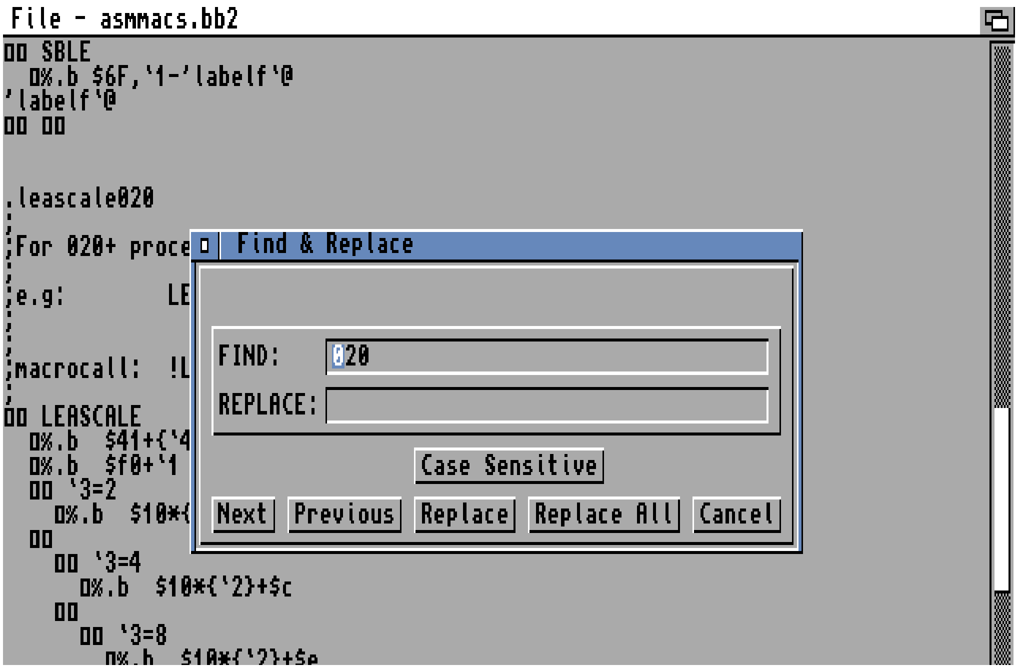
Below, one more "Goto..." option is visible, thanks to which you can place the cursor in a specific program line. This is convenient especially when editing long listings. After selecting this function, a small window will appear on the screen with a text field:



Enter the number of the desired line and press ENTER or select "OK". The program changes the position of the cursor and automatically scrolls the text so that the part indicated in this way is visible on the screen. Of course, using this option does not exclude the "normal" change of the cursor position.

SEARCHING THE TEXT

If you want to find specific text in the whole program, go to the pull-down menu called "Search". The first item is "Find..." and it causes the appearance of a new window. For example, like this:



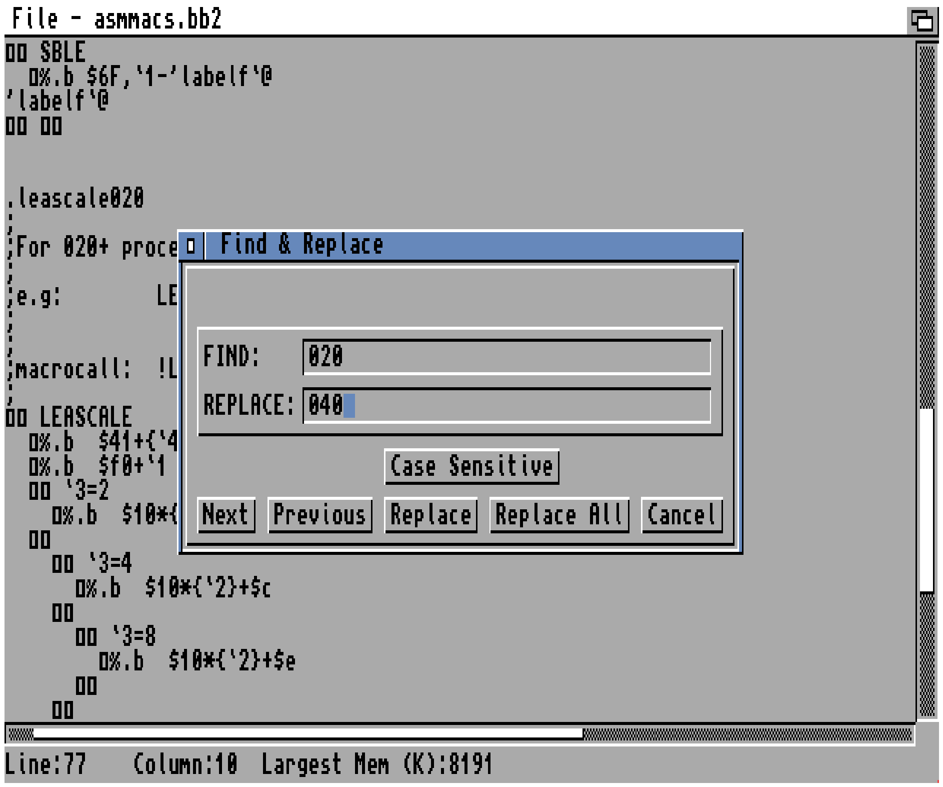
```
□□ SBLE
  □%.b $6F,'1-'labelf'@
'labelf'@
□□ □□


.leascale020
;
;For 020+ proce
;
;e.g:        LE
;
;
;macrocall:  !L
;
□□ LEASCALE
  □%.b  $41+{'4
  □%.b  $f0+'1
  □□ '3=2
    □%.b  $10*{
  □□
    □□ '3=4
      □%.b  $10*{'2}+$c
    □□
      □□ '3=8
        □%.b  $10*{'2}+$e
```

FIND: 020

REPLACE:

In the "FIND" field, enter the text you want to search for. I used the entry "020", because we are interested in finding parts of the program intended for the 68020 processor. Next choose the "Next" button. The program will now be searched from the cursor position "down", which is forward. If you want to get the same in the opposite direction - from the cursor position backward - use the "Previous" field.

The replacement of text fragments works very similarly. To change our entry "020" to "040", you should fill in the field "REPLACE". It should contain the new text to replace the previous one. For example, the whole entry may look like this:



File - asmmacs.bb2

```
□□ SBLE
  □%.b $6F,'1-'labelf'@
'labelf'@
□□ □□


.leascale020
;
;For 020+ proce
;
;e.g:        LE
;
;
;macrocall:  !L
;
□□ LEASCALE
  □%.b  $41+{'4
  □%.b  $f0+'1
  □□ '3=2
    □%.b  $10*{
  □□
    □□ '3=4
      □%.b  $10*{'2}+$c
    □□
      □□ '3=8
        □%.b  $10*{'2}+$e
      □□
    □□
```

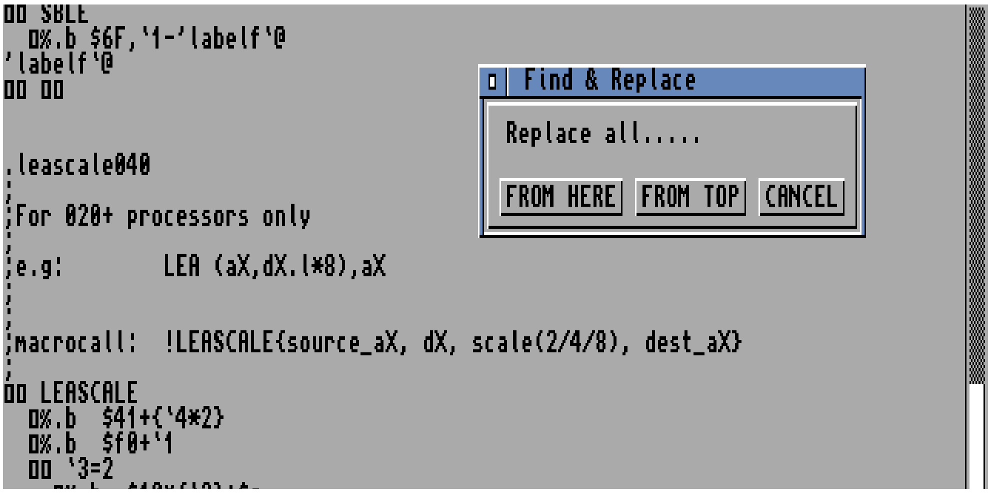
Find & Replace

FIND: 020

REPLACE: 040

Line:77   Column:10 Largest Mem (K):8191

Now use the "Replace" button to find and replace one expression, which is consistent with the "FIND" field. If you want to change all matching parts, use the "Replace All" function. In the latter case, an additional window will appear on the screen as on the next picture:



```
DD SBLE
  DX.b $6F,`1-'labelf`@
'labelf`@
DD DD


.leascale040
;
;For 020+ processors only
;
;e.g:        LEA (aX,dX.l*8),aX
;
;
;macrocall:  !LEASCALE{source_aX, dX, scale(2/4/8), dest_aX}
;
DD LEASCALE
  DX.b  $41+{`4*2}
  DX.b  $f0+`1
  DD `3=2
```

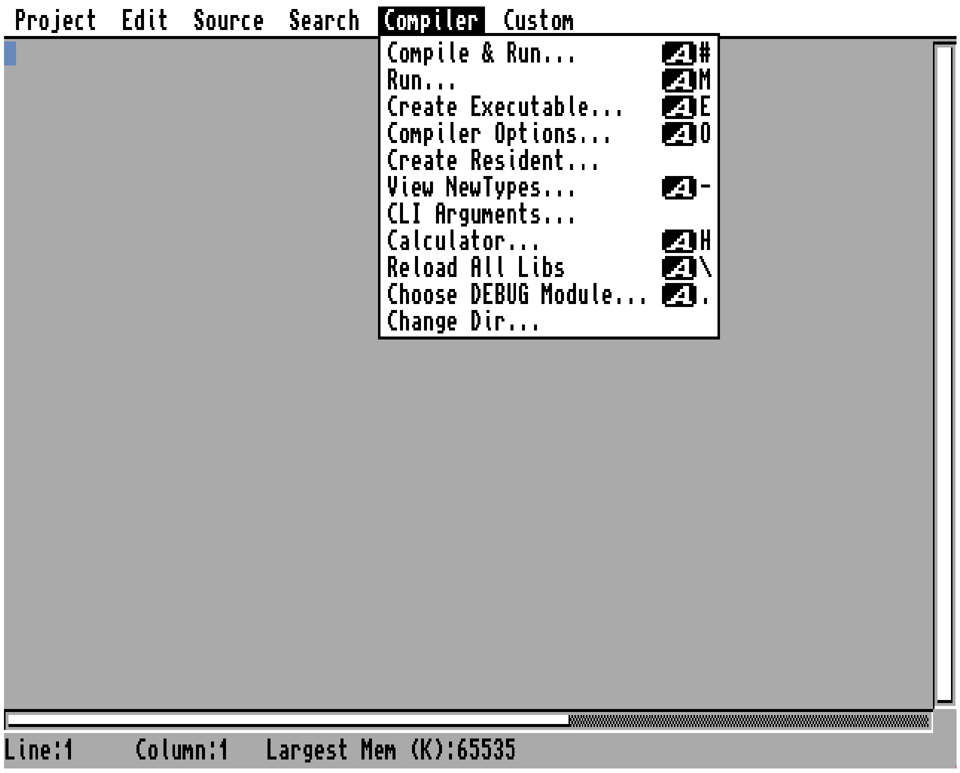
The "FROM HERE" button causes the function works from the current cursor position to the end of the text. The "FROM TOP" means searching the file from the beginning, regardless of the cursor position. If you want to cancel the operation, use the third button labeled "Cancel".

Note that regardless of the option selected, the cursor will always be set at the beginning of the text being searched for or replaced. It will not be highlighted, but you do not have to check manually where the proper text string is located.

The same functions as in the window can also be found in the pull-down menu called "Search". You can use them in the same way, because they work the like the buttons in the window. Thanks to them, you will not change the text you are looking for, but you will start searching or replacing strings faster than in the "Find & Replace" window.

COMPILATION AND TESTING

So far, we have talked about purely editorial functions of "Ted". Using it you will also launch programs, so you need to know the options shown in the pull-down menu named "Compiler". It is much more complicated, so first look at all the items:

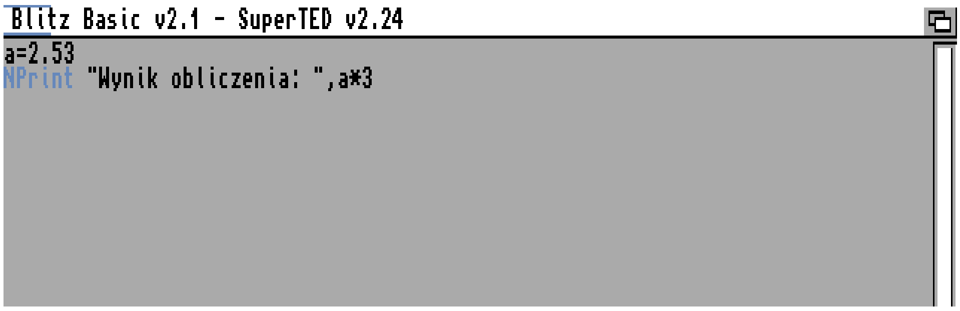


The name of the menu points to the action called "compilation" of the program. It is a process that involves transforming the content visible in the editor into a machine code, that is a set of commands intended for a processor that can already be started. So, as a result, you will get an executable file like any "regular" program for Amiga.

That is why the name of the option we will use is "Compile & Run...", which means that two main activities will be performed: compiling and running the program. Of course, everything works if the listing does not contain errors, otherwise you'll see a message about the problem that needs to be resolved. You can read more about this topic in the chapter titled "Errors handling".

- Starting the program

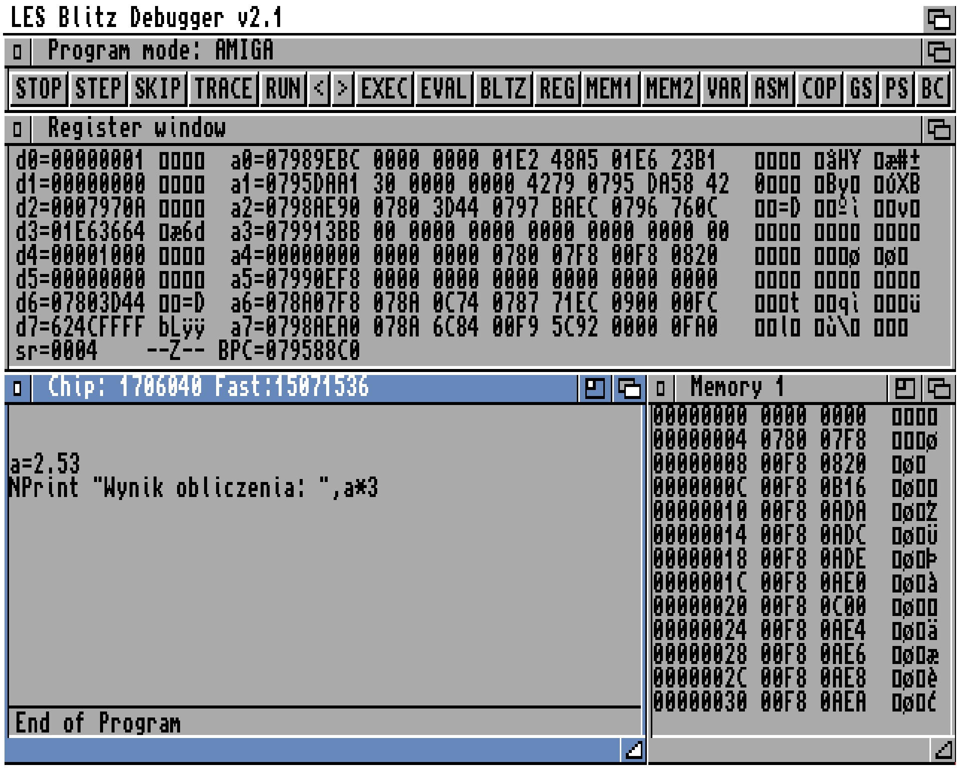
To see how it works in practice, we will try to run a very simple program. Take a look at the example:


Blitz Basic v2.1 - SuperTED v2.24

```
a=2.53
NPrint "Wynik obliczenia: ",a*3
```

These are just two lines - declaration of the variable and displaying the result of calculating the product of two values. We have applied a lines that certainly will not cause you any problems.

Now select the "Compile & Run..." option from the pull-down menu called "Compiler". After a moment the screen will be switched and you will see the content like this:


LES Blitz Debugger v2.1
Program mode: AMIGA
STOP STEP SKIP TRACE RUN < > EXEC EVAL BLTZ REG MEM1 MEM2 VAR ASM COP GS PS BC
Register window
Chip: 1706040 Fast:15071536
Memory 1

```
d0=00000001 □□□□  a0=07989EBC 0000 0000 01E2 48A5 01E6 23B1   □□□□ □ǎHY □æ#±
d1=00000000 □□□□  a1=0795DAA1 30 0000 0000 4279 0795 DA58 42  □□□□ □By□ □óXB
d2=0007970A □□□□  a2=0798AE90 0780 3D44 0797 BAEC 0796 760C   □□=D □□º̀ì □□v□
d3=01E63664 □æ6d  a3=079913BB 00 0000 0000 0000 0000 0000 00  □□□□ □□□□ □□□□
d4=00001000 □□□□  a4=00000000 0000 0000 0780 07F8 00F8 0820   □□□□ □□□ø □ø□
d5=00000000 □□□□  a5=07990EF8 0000 0000 0000 0000 0000 0000   □□□□ □□□□ □□□□
d6=07803D44 □□=D  a6=078A07F8 078A 0C74 0787 71EC 0900 00FC   □□□t □□qì □□□ü
d7=624CFFFF bLÿÿ  a7=0798AEA0 078A 6C84 00F9 5C92 0000 0FA0   □□l□ □ù\□ □□□
sr=0004    --Z-- BPC=079588C0
```

```
a=2.53
NPrint "Wynik obliczenia: ",a*3
```

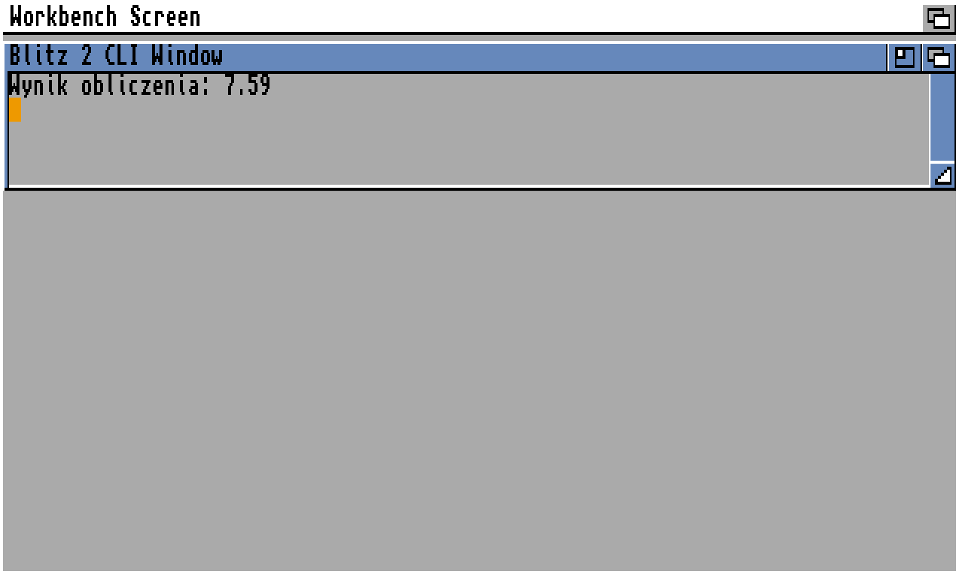
```
End of Program
```

```
00000000 0000 0000  □□□□
00000004 0780 07F8  □□□ø
00000008 00F8 0820  □ø□
0000000C 00F8 0B16  □ø□□
00000010 00F8 0ADA  □ø□Ú
00000014 00F8 0ADC  □ø□Ü
00000018 00F8 0ADE  □ø□Þ
0000001C 00F8 0AE0  □ø□à
00000020 00F8 0C00  □ø□□
00000024 00F8 0AE4  □ø□ä
00000028 00F8 0AE6  □ø□æ
0000002C 00F8 0AE8  □ø□è
00000030 00F8 0AEA  □ø□ê
```

We must admit that the situation has become more complicated. What you see is the so-called "debugger", or module designed to analyze the operation of your program. One of the windows displays the content you know, the second - memory, and the third - the status of so-called "registers" of the processor. These are memory cells that are used to store information, such as calculation results or other temporary data.

A toolbar is available at the top of the screen that you can use to call various functions. We will be able to discuss these elements in detail later, now it is only important to know, how to check the operation of the program.

Contrary to appearances, it is very easy, just switch the screen to the Workbench. You should see the following window:



```
Wynik obliczenia: 7.59
```

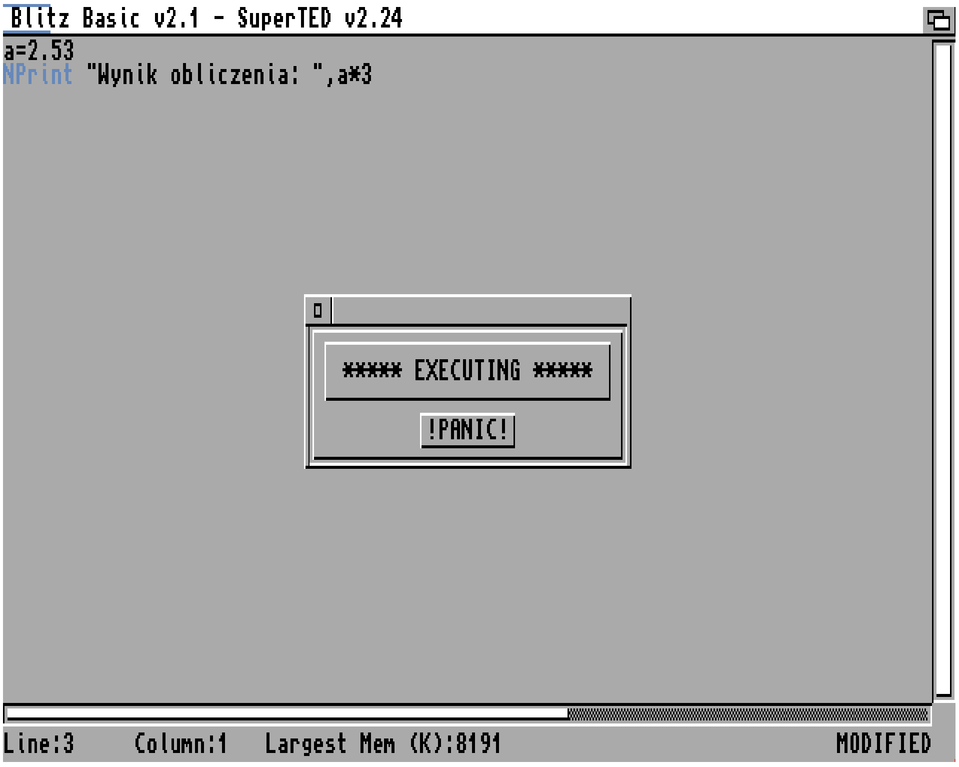
This is the "CLI" window that opens Blitz Basic when the program is started. Here the result of actions is visible. Please note that we have not used commands to open a new screen or window, or create other graphical interface elements.

That is why the result of NPRINT command was shown in the default place - in the AmigaDOS window.

When you switch the screen back to the "Ted", in the central part you will see a window containing the following message:

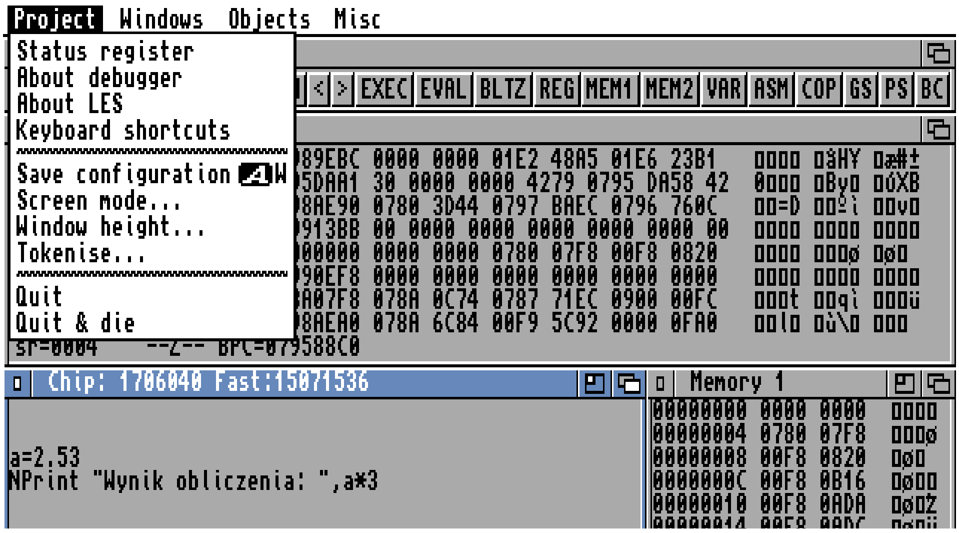
EXECUTING

and the smaller button below. See next picture:



```
Blitz Basic v2.1 - SuperTED v2.24
a=2.53
NPrint "Wynik obliczenia: ",a*3
```

This is because the listing is not formally completed in any way. To stop the operation and return to editing the program, switch the screen again to the "debugger" and call its pull-down menu named "Project". You'll see a list of options:



```
89EBC 0000 0000 01E2 48A5 01E6 23B1
5DAA1 30 0000 0000 4279 0795 DA58 42
8AE90 0780 3D44 0797 BAEC 0796 760C
913BB 00 0000 0000 0000 0000 0000 00
00000 0000 0000 0780 07F8 00F8 0820
90EF8 0000 0000 0000 0000 0000 0000
A07F8 078A 0C74 0787 71EC 0900 00FC
8AEA0 078A 6C84 00F9 5C92 0000 0FA0
sr=0004    --Z-- BPC=079588C0
```

```
a=2.53
NPrint "Wynik obliczenia: ",a*3
```

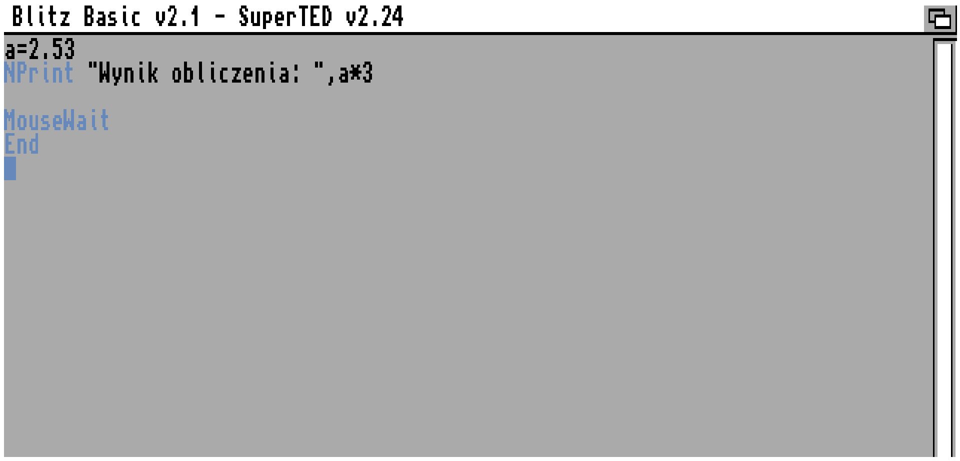
```
00000000 0000 0000
00000004 0780 07F8
00000008 00F8 0820
0000000C 00F8 0B16
00000010 00F8 0ADA
```

Choose "Quit" to continue working on the program. If you want to avoid this problem, all you have to do is put a line with the END command at the end of the program. Thanks to this you will see the result of the action in the "CLI" window, and then Blitz Basic will automatically switch the screen back to the editor.

However in this case you have no control over the time that the "CLI" window will be displayed. It can happen quickly enough that you will not notice the results of the program and the window will be closed. There is a simple solution for this. In the penultimate line, before the END command, enter:

```
MouseWait
```

Now the entry should look similar to the one shown in the next picture:



```
a=2.53
NPrint "Wynik obliczenia: ",a*3

MouseWait
End
```

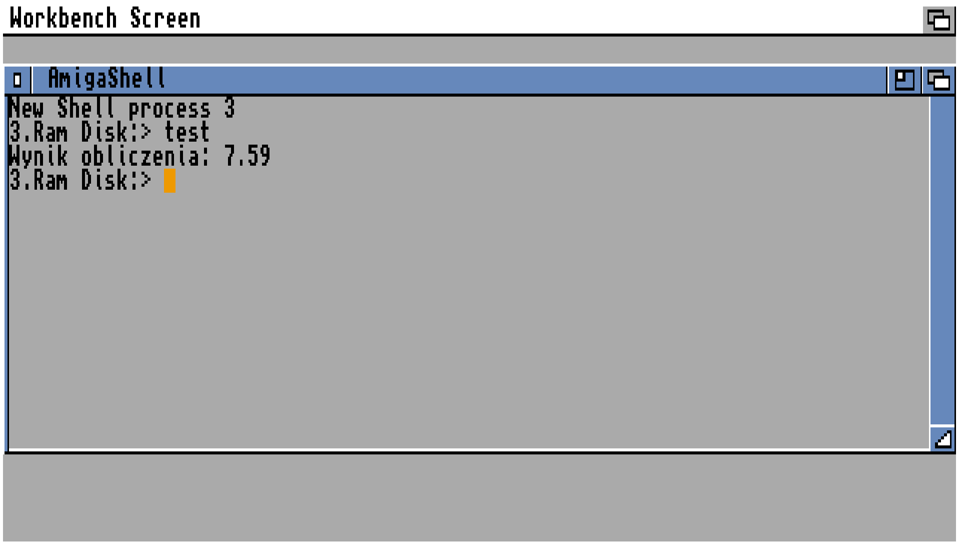
Now the program will wait until you press the left mouse button, then the window will be closed and you will return to the editor. This is a very often used method of testing the operation of programs, but it is worth adding one important note. The MOUSEWAIT command should not be used for other purposes, because it slows down the work of other programs running simultaneously. Use it wherever you want to check the results of the entered commands, but you should delete it from the finished program.

Also remember that correctly entered commands are highlighted in the editor in blue. At the same time, after pressing the ENTER key, the spelling is corrected. Thanks to this, you can be sure that you have not made a mistake and the program becomes more readable. Command names are visually separated from arguments and variables. Of course, this does not affect the operation of the program and this is visible only in the Blitz Basic editor.

- Creating executable files

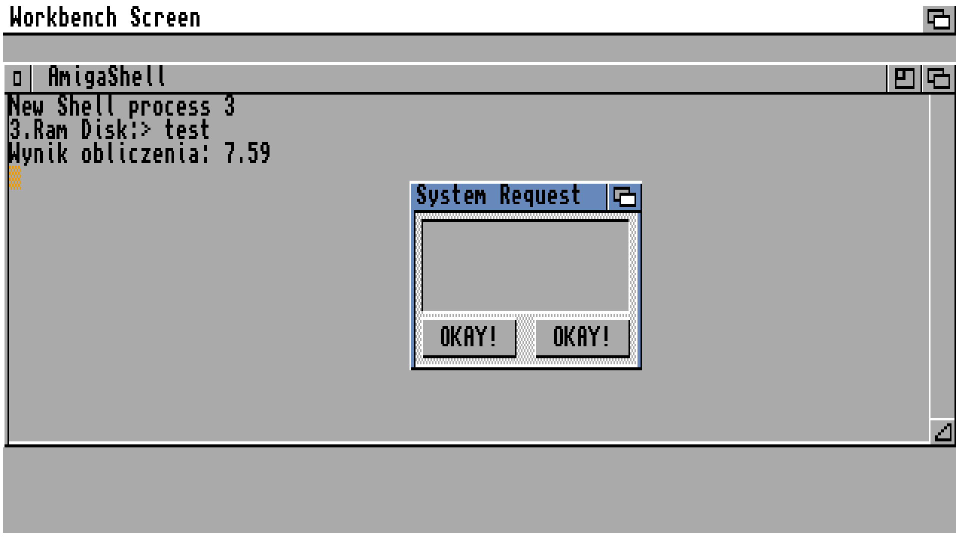
By using the "Compiler" pull-down menu option you can also save the program ready to run outside the editor. Just select the option named "Create Executable...". This way you create a so-called "executable" file, i.e. a program that can be run in the AmigaDOS window.

After choosing this option, a selection window will appear on the screen, and you should select the directory where you want to save the file and enter its name. After using the "Ok" button in the target directory you will see a new item. Now open the "Shell" window and type the file name. On the picture it has the name "test" and the result of work looks as follows:



```
New Shell process 3
3.Ram Disk:> test
Wynik obliczenia: 7.59
3.Ram Disk:>
```

The program works exactly as after launching via the Blitz Basic editor, i.e. displays the message and waits for the left mouse button to be pressed. You can also see a smaller window with the "OKAY" buttons as below:

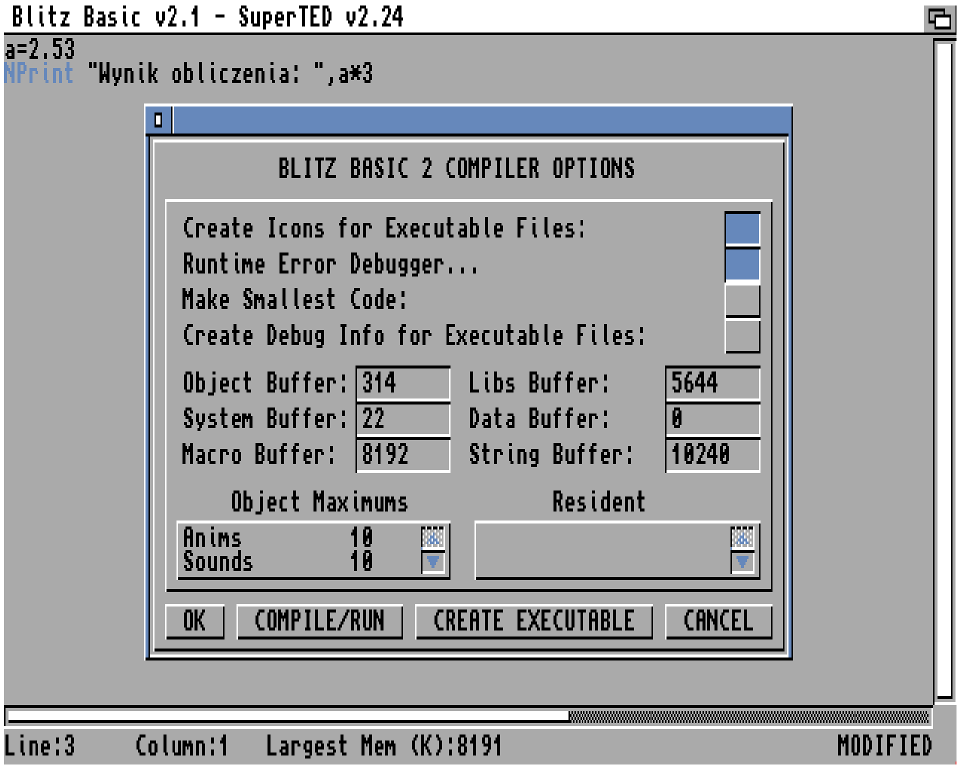


```
New Shell process 3
3.Ram Disk:> test
Wynik obliczenia: 7.59
```

Select one of them to end the program. In this way, you can save your programs. The more advanced use of the functions will be discussed later.

- Compilation parameters

The default compilation takes a specific group of parameters that can not be seen. Sometimes changes will be needed. It is possible using the next option placed in the pull-down menu named "Compiler". This time, select "Compiler Options...". A large window will appear on the screen:


Blitz Basic v2.1 - SuperTED v2.24

```
a=2.53
NPrint "Wynik obliczenia: ",a*3
```

BLITZ BASIC 2 COMPILER OPTIONS

Create Icons for Executable Files: ☑
Runtime Error Debugger... ☑
Make Smallest Code: ☐
Create Debug Info for Executable Files: ☐

Object Buffer:	314	Libs Buffer:	5644
System Buffer:	22	Data Buffer:	0
Macro Buffer:	8192	String Buffer:	10240

Object Maximums

Anims	10
Sounds	10

Resident

Line:3 Column:1 Largest Mem (K):8191 MODIFIED

The first function concerns creating icons for executable files. By default, it is active, so along with your program will be saved file with the extension ".info" in the destination directory. For now, we write programs to run in the AmigaDOS window, so you can disable the "Create Icons for Executable Files" function.

Also note the "Runtime Error Debugger" option. Thanks to it, after launching the program, the "debugger" screen appears. If you do not want to watch it, turn this feature off as well.

However, you must remember that the "debugger" is a module that not only indicates important information about the program, but also helps when an error occurs. If you disable it, your program may hang and you will not see an error message. When you write simple programs, it may not be important, but as you expand the listing, you should use this function.

This may be useful also when you use Amiga with a small amount of memory, because the "debugger" increases memory consumption. In the case of more complex programs, this module will be useful, for example to "track" various commands. We will write more about it later in the book.

Be interested in the function described as "Make Smallest Code". When you turn it on, the compiler will try to always create files with the smallest volume. Again, it will not be very important when running simple listings, but after loading a more expanded program, you will get interesting results.

Results depend on the ways of writing the program, its individual parts and the algorithms used, so saving space will not always be possible.

Other fields in the window should be treated as information only. Below the mentioned buttons you can see the values denoting buffers for various types of operations and variables. At this stage, you do not have to worry about them.

Note that using the buttons located at the bottom of the window, you can run the program or create an executable file without having to call the pull-down menu again. The following fields are used for this purpose:

- COMPILE/RUN,
- CREATE EXECUTABLE.

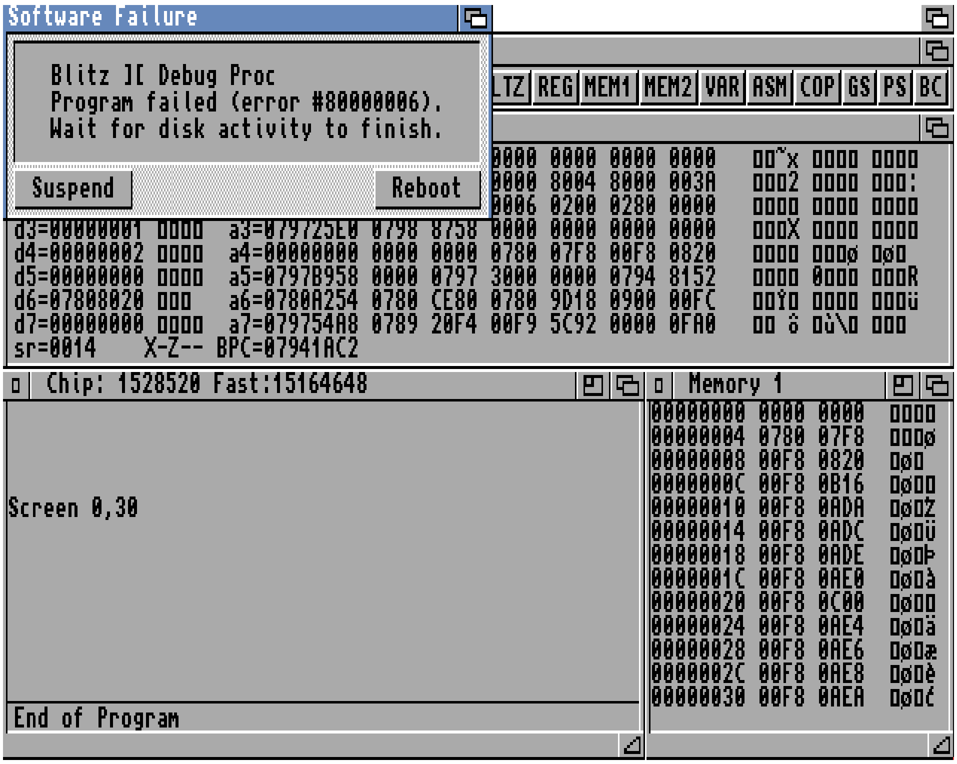
Their operation is identical to the functions of the "Compiler" menu.

If you only want to change the configuration, click "OK" to close the window. When your intention is to skip changes, select "CANCEL". The window works in a similar way to configuration functions that can be found in many other programs.

Compilation and launching the program is a complex process, which we will write about later in the book. At this point, you should be able to compile the program and set the basic options to run.

Remember that Blitz Basic operates on the components of the operating system and in many cases your program may lead to the computer crash or the "Software Failure" window appears. In addition, you should not run the program for the second time without disabling the "debugger" module.

So if you see a window similar to the one below:



```
Blitz ][ Debug Proc
Program failed (error #80000006).
Wait for disk activity to finish.
```

```
                                     0000 0000 0000 0000   □□˜x □□□□ □□□□
                                     0000 8004 8000 003A   □□□2 □□□□ □□□:
                                     0006 0200 0280 0000   □□□□ □□□□ □□□□
d3=00000001 □□□□  a3=079725E0 0798 8758 0000 0000 0000 0000   □□□X □□□□ □□□□
d4=00000002 □□□□  a4=00000000 0000 0000 0780 07F8 00F8 0820   □□□□ □□□ø □ø□
d5=00000000 □□□□  a5=0797B958 0000 0797 3000 0000 0794 8152   □□□□ 0□□□ □□□R
d6=07808020 □□□   a6=0780A254 0780 CE80 0780 9D18 0900 00FC   □□Ŷ□ □□□□ □□□ü
d7=00000000 □□□□  a7=079754A8 0789 20F4 00F9 5C92 0000 0FA0   □□ ô □ù\□ □□□
sr=0014    X-Z-- BPC=07941AC2
```

```
Chip: 1528520 Fast:15164648

Screen 0,30

End of Program
```

```
00000000 0000 0000   □□□□
00000004 0780 07F8   □□□ø
00000008 00F8 0820   □ø□
0000000C 00F8 0B16   □ø□□
00000010 00F8 0ADA   □ø□Ź
00000014 00F8 0ADC   □ø□Ü
00000018 00F8 0ADE   □ø□Þ
0000001C 00F8 0AE0   □ø□à
00000020 00F8 0C00   □ø□□
00000024 00F8 0AE4   □ø□ä
00000028 00F8 0AE6   □ø□æ
0000002C 00F8 0AE8   □ø□è
00000030 00F8 0AEA   □ø□ć
```

it means that you made a mistake and you have to reset the computer. Next, you should specify whether the problem was caused by an incorrect command state in the program or you just incorrectly activated the function available in the editor or the "debugger". To do this, you can create a file ready to run in the AmigaDOS window or change the compiler configuration options.

KEYBOARD SHORTCUTS

Some "Ted" functions may be uncomfortable for calling with the mouse. This is a basic way of handling, but nothing prevents you from using the key combinations. Usually, in programs for Amiga, the options from the pull-down menu can be run using the AMIGA key and the corresponding character. In this case it is the same.

Here is a list of available functions:

AMIGA + A
It calls the file saving function, so you will save the program under a new name. This is equivalent to the "Save As..." option from the "Project" menu.

AMIGA + B
Places the cursor in the last line of the program.

AMIGA + D
Removes the line of text on which the cursor is located.

AMIGA + F
Calls the search function ("Find..." option in the pull-down menu called "Search").

AMIGA + G
Places the cursor in a program line specified by the user.

AMIGA + I
It allows you to load another file and insert its contents at the current cursor position. The other content under the cursor is scrolled down.

AMIGA + J
It allows you to connect two consecutive program lines.

AMIGA + L
It invokes the function of loading a file from the disk, with the loss of the current content.

AMIGA + N
It allows you to search for the next fragment given earlier in the "Find ..." window.

AMIGA + Q
Quits the Blitz Basic editor. You will return to the window or screen where you started the program. Usually it is a Workbench screen or AmigaDOS window.

AMIGA + R
Invokes the function of replacing fragments of two text strings. This is the equivalent of the "Replace" option in the pull-down menu called "Search".

AMIGA + S
It allows you to save the current program content to disk, but under the same name. The exception is the situation when the file

is saved for the first time on the disk. This is the equivalent of the "Save" option from the menu named "Project".

AMIGA + T
Places the cursor at the beginning of the program.

AMIGA + W
"Disables" the highlighting of the selected text fragment when block operations are used. You can also read about it in a separate section.

AMIGA + Y
Removes the line content at the cursor position until it encounters an End-Of-Line character. For this reason, it can delete a larger or smaller part of the program, because the EOL mark is invisible by default.

AMIGA + ?
Invokes the editor information window where we will find, among other things, a message with the name of the so-called public screen on which the program operates. In addition, the ARexx language port will be indicated, if available. Otherwise, the "N/A" (Not Available) symbol will be displayed which means no active ARexx port.

AMIGA +]
Moves the selected block to the right, that is, creates or increases the so-called "indentation". Before using it, you should highlight the block, for example with the mouse.

AMIGA + [

The inverse function of the previous one, it allows to reduce the "indentation" size.

AMIGA + }

It allows you to automatically add a semicolon at the beginning of the lines of the selected block of text. Thanks to this, these lines will be treated as comments and will not be executed when the program is started.

AMIGA + [

Inverse to the previous function, it deletes semicolon characters, which means "reactivation" the lines indicated in the block. The other content is not modified.

THE FIRST PROGRAM

VARIABLES

The programs contain text and numeric data. We can define them as so-called "variables", and also perform many different operations on them. This can be compared to the data we use when solving math problems, substituting specific values in the formulas. Similarly, it will work in your program.

To influence the operation of the program, it is necessary to use many different variables, which will usually take numerical or text values. Blitz Basic supports 5 standard numerical types with different ranges and accuracies. Here is their list:

- Byte	(.b)	+-128	integers
- Word	(.w)	+//-32768	integers
- Long	(.l)	+/-2147483648	integers
- Quick	(.q)	+/-32768.0000	1/65536
- Float	(.f)	+/-9*10/\18	1/10^18

Please note that the use of different types of variables not only changes the accuracy of stored information, but also the speed of operations.

The variable is assigned to a specific type by adding the appropriate suffix to the name. You have to do it only at the beginning, and subsequent references may be less detailed, unless it is a text variable. A special set of variables are so-called "arrays", which you can read about in a separate section under the same title.

Blitz Basic also allows the use of many extensions of functions, among other things, it is possible to define new types of variables that will be a collection of several standard types.

However, it is important to remember that a few basic rules should be followed when creating variables and arrays. Their names may have any length, but they should clearly describe the operation. Too long symbols will be uncomfortable to use. Each name should start with a letter or an underscore character and contain only letters or numbers. In addition, the names can not overlap with the other Blitz Basic commands, because the operation of program would be disrupted.

An interesting fact is that the spelling is also distinguished, i.e. the size of letters. For example, the following two names:

```
pointone
```

and

```
PointOne
```

will be treated as two separate items. Always remember about these rules, otherwise your program will not work properly. In such situations, usually without changing the variable names you will not be able to eliminate all errors.

DECLARING VARIABLES

Each variable has its own symbol and assigned value. To create it, you must enter a line similar to the following:

```
a=10
```

This will cause the numeric variable "a" to contain a value of 10. As you can see, it is not complicated, but each name must be entered very precisely. If your variable is to take non-integers, you must follow the list mentioned in the "Variables" section. For example, after entering a line:

```
a.q=1.3
```

the variable "a.q" will take the number 1.3, so a fraction. This is due to the type of variable we chose by typing the ending ".q". Variables may also contain the result of the calculation:

```
a.q=50/13
```

Now the new variable "a.q" will contain the value 3.8461. This is a convenient way to enter various numbers, because you do not have to manually calculate them, and their precision in the program depend on the type of variable.

Similarly, it is possible to create a variable containing text. In this case, its symbol must be completed with a "dollar" sign ($), so instead of the "a" variable you must use the "a$" entry, for example:

```
a$="Amiga"
```

Now the new text variable "a$" will contain the word "Amiga". Note that in the same way you can assign a number to a text variable:

```
a$="10"
```

However, it will be treated all the time as a text string, so you will not be able to do calculations on it. Of course, you can perform other activities characteristic of text variables. We'll write about it in the next sections.

OPERATIONS ON NUMERIC VARIABLES

You can make the computer change the value of a numeric variable by a specific pattern, for example added or subtracted a specific value. Just use the following line:

```
a=a+1
```

or

```
a=a-1
```

It can also contain more than one operation, you can also use parentheses. Here's another example:

```
a=(a+2)*10
```

Of course, the usual sequence of mathematical operations will be maintained. Please note that after the equality sign, in addition to the formula, we wrote the symbol of variable "a". How does this work? The computer takes into account the previously assigned value and performs the calculation according to the formula. In the case of the first entry, the number 1 is added or subtracted from the current value of the variable "a", and the whole is saved within the same variable. To better understand this operation, you can create a second variable and apply the following entry:

```
b=a+1
```

However, the program should not contain too many variables without a clear need. In addition, such a entry does not make sense if you want to change the values many times, because you would have to create more items each time.

The calculation result may also be dependent on more variables, not just one. The previous formula can be changed into this:

```
c=(a+b)*10
```

You must declare an additional variable "b" beforehand. The entire program can therefore look like this:

```
a=5
b=8

c=(a+b)*10
```

Remember that for now we only specify variable values. How to display them on the screen you will learn, among others, in the section "Commands and syntax".

It is worth adding that the exponentiation is obtained by the sign "^". This is a typical symbol for the Basic language, but keep in mind that in case of complex entries you will often have to use single or double brackets. This may look like this:

```
c=2^(a+b)*((15*b)-5)
```

This is just an example, but shows well how it works. The brackets should be used everywhere where you want to get more legibility related to the order in calculations.

In your programs, you can use fractions by storing them also in decimal form:

```
a=2.36
```

The fraction can be easily rounded if necessary, just use the INT() function. The entire program may look like this:

```
a=2.36
b=Int(a)

NPrint a
NPrint b
```

Please note that we are creating a new variable "b" here, but this is not necessary. If you do not care about keeping both results, you can delete the last line and then change the rounding to the new entry:

```
a=Int(a)
```

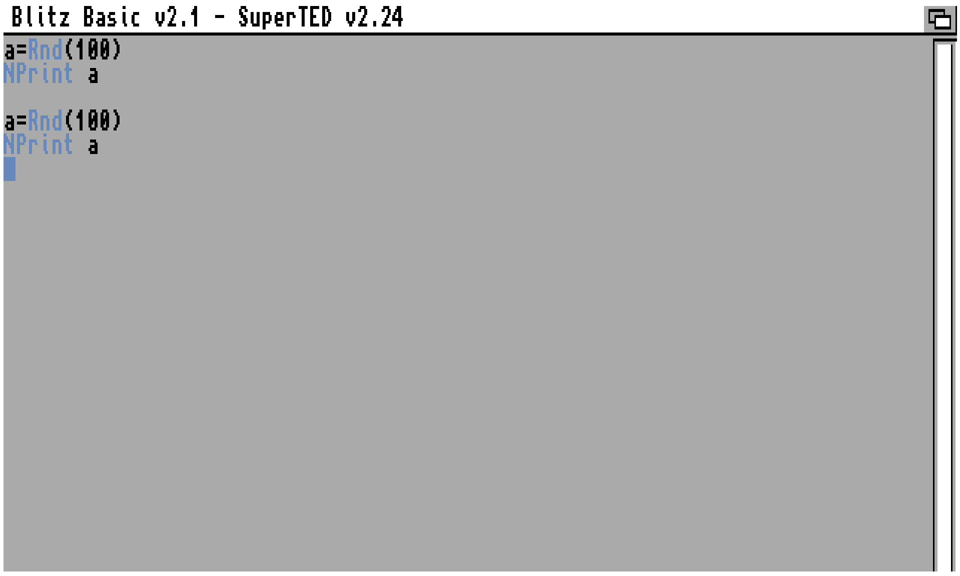
On the screen you will see the same number, that is "2". Remember that the INT function does not work like a typical rounding down or up circle, but just cuts off a fractional value.

If necessary, we can also generate a random number using the RND() function. The range of values should be given as an argument. For example:

```
a=Rnd(100)
```

This will cause the maximum value to be 100, and you can get a different result each time you call a function. This happens regardless of whether you run the program twice or repeat a few lines in one listing and start the program later.

Remember that the number is "created" at the time of executing the line with the saved function name, so it is not enough to enter the NPRINT command twice. The correct example is presented in the next illustration:


Blitz Basic v2.1 - SuperTED v2.24

```
a=Rnd(100)
NPrint a

a=Rnd(100)
NPrint a
```

You can read more about the function and how to use it in a separate section dedicated to this topic.

OPERATIONS ON TEXT VARIABLES

Within the variables you can save any text. You can also perform other types of operations on them, not available for numbers. The simplest way to combine two variables, because we do it identically as for numbers, i.e. for example:

```
a$="Amiga"
b$="1200"

wynik$=a$+b$
```

The only difference is the use of the dollar sign ($). Such a record is universal, which means that you can use it both by giving variable names and entering text directly. It may look like this:

```
wynik$=”Amiga”+b$
```

or

```
wynik$=”Amiga”+”1200”
```

You will not make any calculations on the number saved in this form. Instead, you can convert a text string to a number. To make this possible, you need to use the function named VAL(). Referring to our example, the lines:

```
b$="1200"
b=Val(b$)

NPrint b
```

will cause the variable "b" to contain the number 1200, and you can we check it with a typical NPRINT command. Sometimes you may need to perform a reverse operation, for example by combining two strings of text. The next function STR$() may be used for this purpose. It should be used in the same way, giving the text variable as a parameter. E.g:

```
b=1200
b$=Str$(b)

NPrint b$
```

I would like to emphasize once again that variables subjected to operations must have the same type. Therefore, adjusting their type "in both directions" can be very useful, although initially you may not see much sense in these functions.

You can also modify any text variable so that the text is written in small letters or in capital letters. Functions with the following names are used this purpose:

```
UCase$()
```

and

```
LCase$()
```

We use them in the same way, giving the variable name in brackets. For example, in order for "Amiga" inscription to be written in CAPITAL letters, you must use such a program:

```
a$="Amiga"
b$=UCase(a$)

NPrint b$
```

The function names come from the English words "Lowercase" and "Uppercase", which should make it easier for you to remember how they work.

Text strings can also duplicate. If you want to create a second variable containing a certain number of repetitions of the word "Amiga", just enter lines similar to the following:

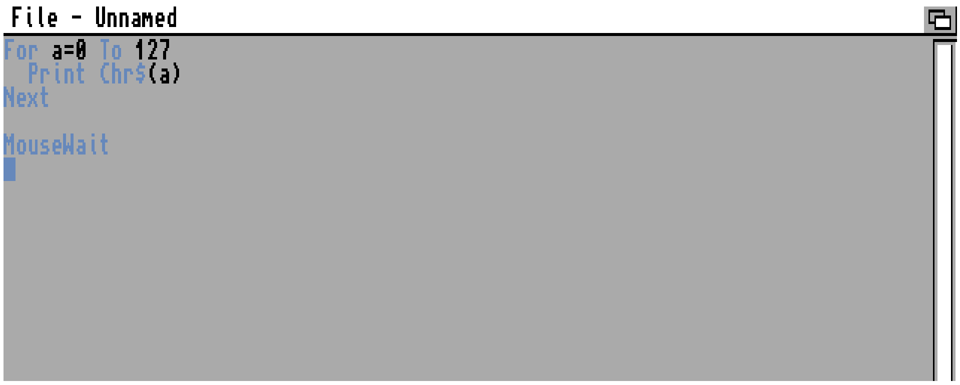
```
a$=″Amiga″
b$=String$(a$,5)

NPrint b$
```

As you can easily guess, the first parameter of the function STRING$() is the name of the source variable, and the second - the number of repetitions. Note that there will be no gap between copies of the text. Sometimes you may want to get such an effect, but most often, we need to keep the text in readable form.

You can place a SPACE sign at the end of the word or a second variable containing a separator. Then combine the variables before doing the duplicate.

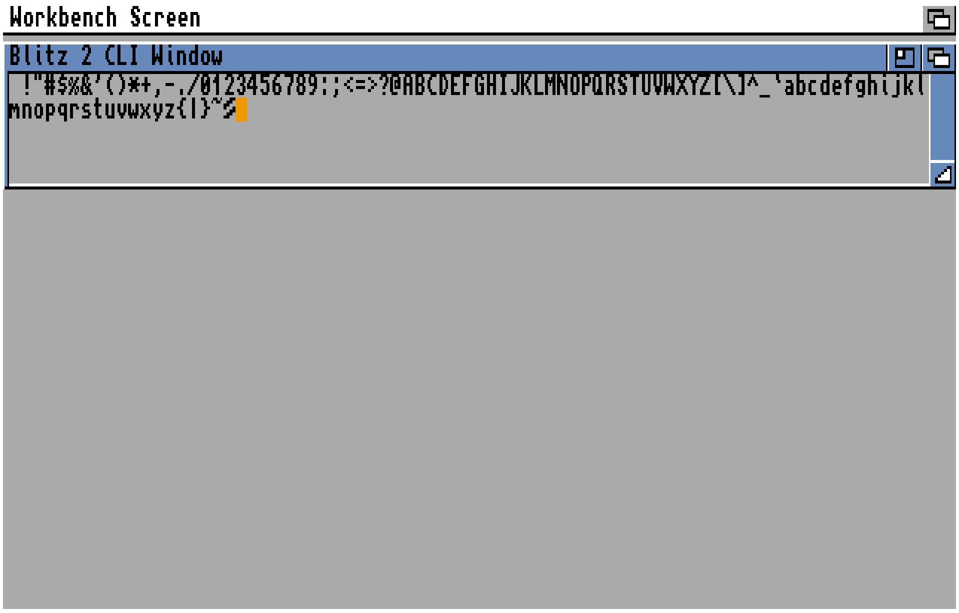
However, entering the spaces manually is not convenient, and you will not be able to automatically change the gap character unless you type it again from the keyboard. To make your task easier and make your program more universal, we suggest learning about the next CHR$() function. With it, you can save a character having a specific value from the so-called ASCII table. It is a code that assigns the numbers to letters of the alphabet, digits and other characters that are not accessible directly from the keyboard. The latter feature is very important if you do not want to impose restrictions on your program.

Of course, now you should check what numbers are assigned to specific characters. That is why we will write a short program that realizes this function. We will use the PRINT command, the FOR ... NEXT loop and declarations of several variables. Here's our next suggestion:


File - Unnamed

```
For a=0 To 127
  Print Chr$(a)
Next

MouseWait
```

See how the result of this listing looks like:



```
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
mnopqrstuvwxyz{|}~
```

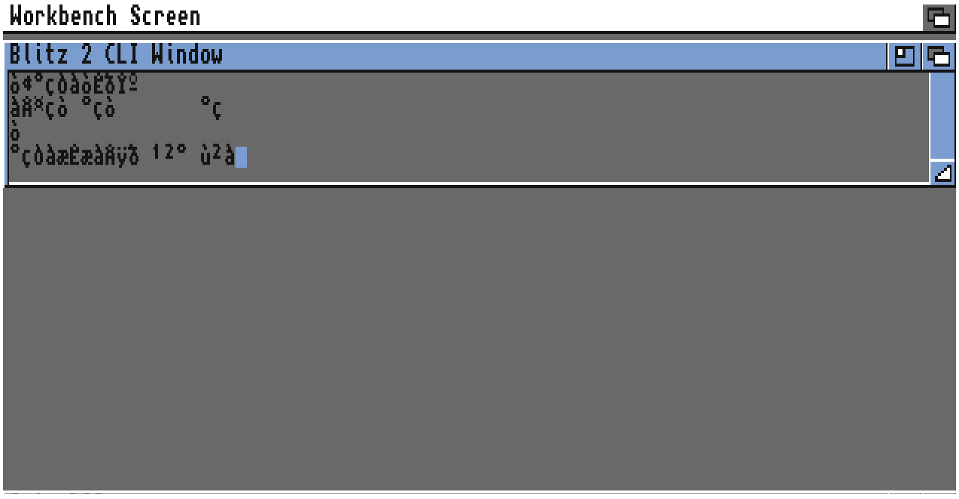
Now you can easily apply the appropriate characters to insert the separator in the middle of the text. You can do it in the following way:

```
a$="Amiga"+Chr$(32)
```

Such line will cause that after the name "Amiga" there will be a space without any additional characters.

Operations on text variables can be even more complicated. What should we do if we create a long text and it will be saved as one item? This can happen, for example, when reading data from a disk. How to do it, read under the section "File Handling".

The result may be a variable containing a text file fragment. After loading the content into the variable and displaying its contents, you will see an unclear form. In an extreme case, you can see something like:



```
ò¢°ÇÒàòÈðÏº
àÅ¤Çò °Çò        °Ç
ò
°ÇÒàæÊæàÅÿð ¹²° ù²à
```

In this situation, the best solution is to read short fragments of the file and analyze the content. To group information you will need to cut the variable into more understandable parts. Later you can, for example, sort commands, detect their arguments and create a program analyzing the operation of AmigaDOS commands. Of course, this is just an example, there are plenty of possibilities to apply.

It does not change the fact that you must in some way divide the variable's content into many smaller parts. The easiest way to do this is by using the arrays that we write about in a separate section. When you declare them you need to know how to "extract" parts of the text. You will find here the functions with the following names:

- MID$(),
- LEFT$(),
- RIGHT$(),
- REPLACE$()

and

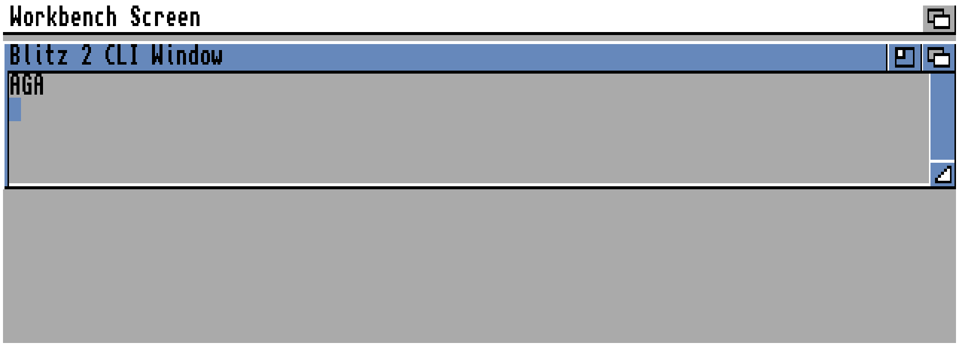
- LEN().

The last one is given separately because it has a different character from the others. All four allow you to perform operations to cut fragments from text variables, though in different ways. The most universal option is to select characters with specific numbers or from a specific range. This is the feature of the first MID$() function.

Just enter the variable name, the first character and the length you want to get. For example, if you enter such a program:

```
a$=″Amiga AGA″
b$=Mid$(a$,7,3)

NPrint b$
```

then you will see that the new variable will contain the text as in the following illustration:



```
AGA
```

You can specify function withous the last parameter, but then you will get a fragment from the character with a specific number to the end of the entire text.

You can do the same, but always starting from the beginning or end of the variable content. In other words, choosing characters will take place from the "left" or "right" side. Therefore, the functions that call these options have the names LEFT$ or RIGHT$. However, in this case you should give only the length of the text you want to get. It may look like this:

```
b$=Left$(a$,5)
```

or

```
b$=Right$(a$,4)
```

In this way, you can divide the variable into many smaller fragments that you will be further processed. These lines entered separately will not cause the effect you want to achieve.

However you can use a loop that will check the successive characters of a variable, then add the conditional statement causes division by specific lengths.

To make this possible, you should learn how to calculate the variable length. This is not difficult, but requires the insertion of another function called LEN(). I placed it at the end of our list, because it does not change the text, but only indicates the length. The usage is as follows:

```
a=Len(a$)
```

As a result, you will receive a number that you can use to check the length of variables in relation to specific characters and to divide the content according to them. If some of the parts do not match the scheme located within the loop, you can change the characters automatically.

Just use the last function from the list, that is REPLACE$(). Our program can now look like this:

```
a$="Amiga AGA"
b$="AGA"
c$="ECS"

d$=Replace(a$,b$,c$)
NPrint b$
```

In this way, the words "Amiga AGA" will be converted into "Amiga ECS". This is because this time the parameters should be given in the following order:

- the string we want to change	(here – a$),
- search string	(here – b$),
- destination string content	(here – c$).

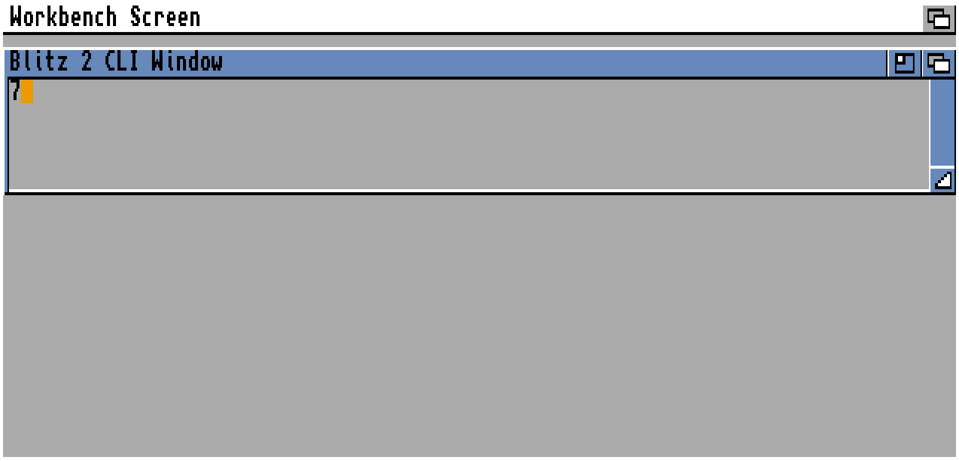
If you enter text as the last parameter that does not exist in the source variable, then no change will be made. Note that anyway, individual functions in the loop will still be executed - assigning a variable and calling the function REPLACE$(). If you're only interested in replacing a part of the string, you can limit your work to this function.

However, you can not check if the text has been found or not. In addition, the program will not run too fast. You can speed it up by checking whether the variable contains the text you are looking for, and then perform the function. Otherwise, the lines which cause string replacement can be omitted.

To get this effect you must use the function named INSTR(). Its operation is more difficult to understand, because as a result it gives a number, although it still operates on text variables. Now see the next program:

```
a$="Amiga AGA"
b$="AGA"

c=Instr(a$,b$)
NPrint c
```

When you start it, you'll see the result:



```
7
```

It may be strange for you because you will see the number. This is how the function INSTR()works - it gives the position of the first character that matches the search string. In our case it is the number 7 because the "AGA" part begins with the seventh sign of the variable "a$".

This function behaves slightly differently when the searched fragment is not found. You will get a logical value of 0 (zero), which means false, and so the text you are not interested in. More about logical operations you can read, among other, in the section entitled "Operators".

We have not discussed the problem of letter case so far. The REPLACE$() and INSTR() functions can take spelling or not. By default, it is checked, but it will not always be convenient. The CASESENSE command is responsible for checking the uppercase and lowercase letters. If you want to disable it, use this line:

```
CaseSense Off
```

Of course, later it is possible to re-enable the spelling control with the ON argument:

```
CaseSense On
```

Remember that the change occurs when the line is executed. The spelling checker will not be restored to its original state after one of the functions. Both "switching on" and "switching off" the spelling control you must set manually, taking into account the needs and - in particular - the order of the loops and arrays used.

OPERATORS

When comparing the contents of different variables, entering conditional statements and other operations, you can use characters such as "plus" or "minus". These symbols are called operators. They can be divided into many types, for example:

- arithmetic operators,
- operators used to declare variables,
- comparison operators,
- logical operators

and other. In the entire book, you will often be able to find examples and discuss their actions. Below is a list of the most important operators that you can use in the editor.

The first group is the usual mathematical operations that do not require a comment:

+	- adding
−	- subtraction
*	- multiplication
/	- dividing
^	- exponentiation

The next symbols refer to so-called logic functions, the result of which is the value TRUE or FALSE. In the first case, it means -1, while in the second you will get 0 (zero).

Basic logic functions can be written using the following characters:

&	- logical AND function
|	- logical OR function

The AND function returns the TRUE value if two variables participating in the operation also have the value TRUE. The OR function is different, because it is enough for one of the variables to point to TRUE, so that the whole function will return such a value. More on this subject we will also write in further chapters.

You can also use so-called comparison operators, which are a permanent element of conditional instructions. Most are saved as standard, that is:

=	- means equality of variables
<>	- means inequality of variables
<	- means a lower value of the variable,
>	- means higher value of the variable

However, sometimes we want to use the "=" and "<" or ">" signs simultaneously. In this case, the equals sign should be written at the end, that is:

<=	- variable equal to or lower,
>=	- variable equal or higher.

These are not all possibilities, but used most often. Examples of the use of the above symbols can be found in the entire contents of the book. To better understand their operation, go to "Conditional statements" sections.

ARRAYS

In your programs, you may need to manipulate groups of numbers or texts that are related to each other in a way. Such operations enable so-called "arrays".

Imagine a situation in which you want to create variables containing 10 numbers defining the same feature, for example horizontal coordinates on the screen. In the usual way you would have to use 10 different variables. It would not be convenient to create them, not to mention change their values.

Thanks to the arrays, you can put everything into one variable, which is supplemented by the index number, i.e. the number in parentheses. So, instead of the "a" variable you can use these items:

```
a(1)
a(2)
a(3)
a(4)
a(5)
a(6)
...
a(10)
```

Each item is independent and can accept any values, but you can declare them and change their contents a lot easier. Certainly in many cases you will want to apply the loops we write about at the section under the same title.

Defining arrays looks a bit different than declaring ordinary variables. At first, you need to specify how many index entries an array can accept. To do this, we use the word DIM by giving the variable name and the maximum index number. E.g:

```
Dim a(10)
```

This means that the variable "a" will be able to have 11 indexes, because they are counted from zero. Assigning specific values works similarly to creating ordinary variables. You just have to remember to enter each index in brackets after the variable name. See more examples below:

```
a(1)=5
a(2)=6
a(4)=14
a(9)=22
```

You can do calculations on the arrays, just as we've discussed before. Entering patterns does not change, but lines may look quite exotic. For example, to add the values of variables from one array, a formula similar to the following should be used:

```
a(5)=a(1)+a(2)
```

or

```
a(1)=a(1)+1
```

As you can see, the general principle does not change. The formulas may have different variables, and it is also possible to add specific values to only one item. Thanks to their extensive form, arrays can have many more complex ways of using, which we will discuss later in the book.

FUNCTIONS

In Blitz Basic, as in any programming language, you can use mathematical functions. They can take many forms and have many applications. At the beginning it is worth learning to use them. Usually, after a function name, we write a value or several values, just like the command name. However, in this case they always must be enclosed in brackets.

The general scheme for typing each function looks like this:

```
FUNCTION()
```

It is a universal method used on various computers and programming languages. Therefore, if you want to calculate the value of the SINUS function, you must enter the following line:

```
Sin(90)
```

We use the function SIN() (that is Sine), and as an argument we give the number 90. A direct record is not very convenient if you are writing a larger program, therefore the function should be assigned to a variable. We do it the same way as before, for example:

```
a=Sin(90)
```

or in a more complex form:

```
a=Sin(x*t)
```

Please note that as part of the function formula, it is possible to use other variables. You can also use other types of functions, not just purely mathematical. They let you get a lot of different information, for example with use of the following function:

```
Asc()
```

you check the numerical value assigned to a particular character. Some functions read information about your computer or perform operations on variables. The functions are a broad issue, which is why we will introduce them gradually. However, their common feature is specific way of typing lines and necessity of using variables where values are stored, i.e. the results of the function.

COMMANDS AND SYNTAX

In each program you will use many commands, i.e. words that call a specific operation. Each of them must be entered in the proper way. This method of writing is called syntax, because it requires the proper order of words, arguments and other elements necessary to operate. Commands have different syntax, they also require different types of arguments, for example text or numbers.

The simplest command is the word PRINT, which displays information on the screen. In the program, you can do it in the following way:

```
Print ″Amiga″
```

or

```
a$=″Amiga″
Print a$
```

As you can see, only one argument is required here. It can be entered directly next to the command name, you can also use previously learned variables. Of course, the lines can be much more complex. Look at some examples:

```
Print "Witaj",a$
Print a$(i)
Print 3,"CARS",a,a*7+3
```

The most useful command for displaying text is not PRINT, but NPRINT. The difference is that every information will be on a separate line, because the program will automatically add a End-Of-Line character. In the case of the word PRINT, subsequent values will appear side by side, which can also be useful. It's worth knowing the difference.

In the further part of the book I will write about many different operations you can call. For now, it is important that you learn to recognize individual parts of the above lines. Please note that all arguments are separated by a comma character, while text strings are always entered in quotation marks. The argument can be both text, number, array, as well as the entire expression.

Regardless of the function of a specific command, the usage pattern is always strictly defined and must be followed. Otherwise your program will not work or perform unforeseen operations.

Also note that until now, all the commands have been entered into separate lines. You can combine them by using the colon character. So instead of writing:

```
a=1
a=a+1

NPrint a
```

simply write:

```
a=1:a=a+1:NPrint a
```

However, this method is not recommended, because it makes listing analysis difficult. In addition, it does not allow the use of spaces or empty lines that may be useful when expanding the program. Despite this, the effect will be the same, so much depends on your needs and habits.

CONDITIONAL STATEMENTS

In your programs you will have to make the activity dependent on individual elements, for example the state of variables or the choice made by the user. This is possible using the so-called conditional statements.

You have to give the "condition" and operations that will be performed if the condition is true. Otherwise, they will be skipped or another operation will be started.

IF ... THEN

The condition can be entered in one line. The IF ... THEN construction is used for this. After the first word, we enter the condition, and after the second word there must be a command to execute. For example:

```
If b=30 Then Print a
```

The above line will print the contents of the variable "a" with the command PRINT provided that the variable "b" reaches the value 30.

You can place more similar lines, one above the other or in certain parts of the program to get a different action when changing the values of variables.

IF ... ENDIF

In many cases you will need to make more complicated actions that can not be written in one line. Therefore, the condition can be entered in the form of multiple lines.

To change our previous example in this way, use the words IF and ENDIF. See how it can look like:

```
If b=30
     Print a
EndIf
```

For now, the operation will be identical. How profit do we have? Between the beginning and ending lines, we can place any other commands, in many lines. Thanks to this, after fulfilling the condition, the program can perform a more complicated operation.

This structure gives you an additional profit which is the possibility of saving further conditions within one IF ... ENDIF design. Schematically, we can do it like this:

```
If CONDITION
     COMMAND1
     COMMAND2
     COMMAND3
          While CONDITION
               If CONDITION
                    LOOP
```

```
        Endif
                    Wend
        Endif
```

Of course, this is just an example, but it shows how complicated we can have a condition, which was initially introduced as one simple line.

Saving consecutive conditional statements inside the other is called "nesting" and, contrary to appearances, it is not rarely used. As you gradually expand your program, it will certainly become more complex not only in terms of the number of functions. It may turn out that the operation of the program will be dependent on more and more precise values of various variables. If one of them has to reach numbers in certain ranges, you can create many more conditions or use an entry similar to our example.

It is often easier to analyze, and the program will be shorter than entering new IF ... ENDIF lines and repeating only some commands. Remember that you should not complicate the listing without the need.

IF … THEN … ELSE

If the condition stored in the IF line is not true, the program will proceed to the next lines or it will be terminated. However you can make another statement automatically called. The word ELSE is used for this purpose, which must be added at the end of the line. Here is the next example:

```
If a=3 Then Print "Amiga" Else Print "Atari"
```

Everything is still written in one line, but now if the condition is true, a different string will be displayed than if the condition is false. I must admit that this entry is short, but not very convenient and does not give you many possibilities for further action. That's why you can enter the same commands in a different way, using the word ENDIF:

```
If a=3
     Print "Amiga"
Else
     Print "Atari"
EndIf
```

The operation will be identical, but now - as before - you can enter a larger number of lines and call for more extensive functions. Remember that in this case two main parts will be distinguished:

1) from the word IF to ELSE,
2) from ELSE to the ending ENDIF.

All lines placed in this structure will be treated as intended for execution after the condition (part 1) has been true and then, if the condition is false (part 2). Analyzing such a program is more difficult, but it gives new possibilities, because within each of these parts you can place more loops, conditions and other necessary elements.

So we can say that the word ELSE is a supplement and causes execution of another condition, opposite to the first one. You can get a similar effect simply by entering two conditions using the type structure

IF ... ENDIF. This is more complicated, especially when your conditional statements are expanded and you write a lot of them. Therefore, check the operation of conditions on several examples before you decide to use more complex structures. Associations of variable values can be difficult to handle, even if your program is not too long.

Remember also that in some cases you will have to reset values, otherwise the conditions can always or never be true. Variables values do not change automatically after an operation typed as part of a conditional statement. You must take care of it yourself, unless the numbers and texts assigned to them are needed in the further parts of the program.

LOOPS

If you want to do the same operation many times, you can use so-called "loops". They allow you to shorten the program, and in many cases also speed up its operation. You can repeat the same parts of the program, however it does not mean that each time the loop is looped, the result must be the same.

There are several types of loops, in addition within them you can change the values of variables, and thus modify parameters during the operation. At first, remember that each loop executions maintains the same principle of operation.

FOR ... TO ... NEXT

The simplest type of loop causes the execution of commands placed between a few words, without setting additional conditions. The loop only requires the entering of a range of action and a variable that will act as a "counter". For example, using the following line, we can execute the loop hundred times:

```
For i=0 To 100
```

As you can see the integers from 0 to 100 will be counted, and the control variable has the symbol "i". However, this line will not work separately, because you have to enter its end, and you should do it using the word NEXT. The whole can take the form like this:

```
For i=0 To 100
      ...
Next
```

Of course, in place of the dots a fragment of the program should be placed, which will be repeated. The loop cycle can be executed hundred times with many similar expressions, but they can be typed in a different way. Take a look at two more examples:

```
For abc=0 To 100
      ...
Next
```

or

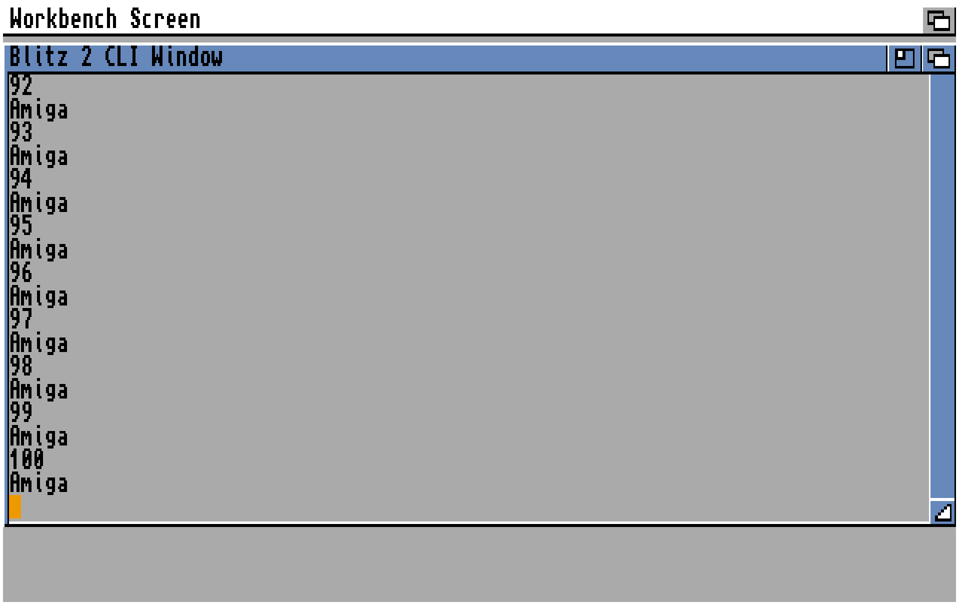
```
For p=514 To 614
      ...
Next
```

In the first case, we only changed the variable name, so it's easy to understand. The second example shows that the loop execution may look the same, despite giving other variable values. Instead of using numbers from 1 to 100, the counter can have any other values.

Note that the variable creating the counter can be used analogously to other variables. Therefore, the repeated fragment inside the loop can use the values of variables like "i", "abc" or "p", although of course it does not have to. The values assigned for these variables do not matter until you associate them with the commands placed within the loop. To better understand this, we will follow the operation of the following program:

```
For abc=0 To 100
      NPrint abc
      NPrint „Amiga”
Next
```

As part of the loop, we put the NPRINT command, which causes the display of new message - always on the new line. More information about displaying text you can read in the section entitled "Commands and syntax". The above entry will cause that the next values of the "abc" variable will appear. The second line is the command to display the word "Amiga", which will be executed each time after displaying the variable value. As a result, you'll get an effect similar to this:



```
92
Amiga
93
Amiga
94
Amiga
95
Amiga
96
Amiga
97
Amiga
98
Amiga
99
Amiga
100
Amiga
```

Of course, the last value will be 100. Now imagine that you only want to display "Amiga" text a hundred times. In this case, it will not matter what values the "abc" variable will take. However, if it is important to place a symbol in the formula, you must remember that the result of the loop will be directly related to the numbers that the loop will generate.

Another important issue is the way the variable value grows. TThe counter does not have to take on the value higher by one, but you can control its increase. The word STEP is used for this, and it must be placed at the end of the line containing the FOR command. Next, we write the value by which the counter should increase instead of "1".

If we modify the previous example, it can take the following form:

```
For abc=1 To 100 Step 2
      NPrint abc
      NPrint „Amiga"
Next
```

Now the counter will change from 1 to 100, but the single step will be 2. In result you will get only 50 loop executions, because the scheme will look like this:

```
1
Amiga
3
Amiga
5
```

```
Amiga
…
```

You can also count back using the same tips. Similarly, the word STEP should be given, but this time with the negative number, for example:

```
For abc=100 To 1 Step -2
```

If you mistake values, then the loop will not be able to be executed, but you will see no error. This can be done by entering "step" as a positive number:

```
For abc=100 To 1 Step 2
```

Keep this in mind because you may be surprised that the program does not work. In this situation, you need to follow the values of the variables.

Remember that the number of loop executions is directly dependent on the numbers that have been saved from in line containing the word FOR. The other values within the loop will only change if a "counting" is made at least once.

The FOR ... NEXT loop will not always be useful, because it performs operations regardless of the content of lines placed between the starting and the ending line. In addition, it always introduces a new variable, which increases the memory requirements. When you create one loop it will not matter. However, when your program will become complex, it may turn out that every kilobyte will be very valuable.

In addition, in many cases you will need to make the loop depend on the content of various variables. This can be achieved by means of so-called conditional statements, which I discuss about in a separate section. Alternatively, it is possible to use other types of loops that do not require the entry of separate lines defining the range of the activity.

WHILE … WEND

If your loop is to be executed until the specific condition ceases to be true, use the WHILE ... WEND construct. Such a loop should be entered in the same way as the previous one, but a condition should be provided next to the first word. E.g:

```
While a<3
      NPrint „Number lower than 3″
Wend
```

As you can guess, the text located next to the NPRINT command will be displayed all the time, unless the variable "a" gets a value higher than 3. For your own use, you can describe it as follows:

make a loop while the condition a<3 IS TRUE

However, this program will not work yet, because the variable has a zero value. We must add the lines declaring the initial value and the scheme of their changes.

It may look like this:

```
a=1

While a<3
      NPrint „Number lower than 3″
      a=a+1
Wend
```

Now the variable "a" will get the value 1, and then - at each loop execution - it will also change by 1. At the fourth cycle, the variable will reach the value 4 and the loop will be terminated.

In the same way you will create more complicated conditions, unless you can not put them in one line, next to the word WHILE. For example, you can associate an activity of your loop with two variable values, but this requires entering the operator in the form of the word AND or OR. What is the difference? Take a look at another example:

```
a=1
b=20

While a<3 AND b>10
      NPrint a
      a=a+1

      NPrint b
      b=b-2
Wend
```

The values of both variables will now be displayed, but the first will increase, the second will decrease. A condition with the word AND means that both variables will be taken into account, although each in a different way. In other words, both conditions will be used simultaneously, all the time given in one line. Our condition can be described in the following words:

make a loop while the variable "a" is lower than 3
AND the variable "b" is higher than 10

The values will change again and the loop will be interrupted only if the values of both variables will be matched the given condition. You can write the same program in another way so that the first line will contain the word OR, that is:

```
While a<3 OR b>10
     …
Wend
```

Such a loop will be executed until one or the other condition is true, but not both at the same time. Let's try to describe it in words once again:

make a loop while the variable "a" is lower than 3
OR the variable "b" is higher than 10

Please note that this completely changes the way the loop works. For operation you no longer need the variable "a" and the variable "b", because only one of them is enough. Thanks to this, the second value can be modified in any way and it will not interrupt the operations placed in the lines between WHILE and WEND words.

Similarly, you can build further conditions where both the words AND and OR will occur. Variables do not have to be numeric, although using them is the easiest way to understand the operation of conditions. E.g:

```
While a<3 AND b>10 OR c$="Amiga"
```

We introduce here another variable, this time containing a text string. The loop requires the presence of two variables or only one, so your program can change the action depending on your needs and purpose of your listing.

Complicated conditions are useful when you want to get the shortest possible program. You do not need to write further, similar parts, but only skillfully set variable values. Maintaining order is all the more important the longer the program is. It is easy to make so-called logic error, that is, to enter a program that works, but does not perform the operation as it appeared in the assumptions. You can read more about this topic under the chapter "Error handling".

REPEAT ... UNTIL

The next variable is similar to the previous one, but the condition should be entered in the last line. Let's modify our previous example:

```
Repeat
     …
Until  a=3
```

The difference is not just about using other words. In this case, the condition will be checked after each loop execution, and not before it is executed. Thanks to this, the REPEAT ... UNTIL loop is always performed at least once.

For simplicity, let us describe the operation in words:

make a loop until variable "a" reaches value of 3

In this way, similarly working parts of the program can be given in various ways, often using a simpler or shorter form. Other options for determining conditions do not change. But remember that the same conditions will work differently, depending on the use of a particular type of loop.

LABELS

You can mark separate parts of the program using so-called labels. To do this, just enter the name that will be assigned to a specific part of the program and end it with a colon. The whole must be written in a separate line, for example as below:

```
point1:
```

This can be useful when analyzing the program, but not only. Blitz Basic will treat this place as a "bookmark", which you can later access using the GOTO command. Just enter the line like this:

```
Goto point1
```

to make a "jump". Please note that this time we do not end the name with a colon. You do not need to give any additional arguments either. The program can have many different labels, and individual fragments are not terminated in any way. We can also write it in the another form:

```
Label1:
      COMMAND1
      COMMAND2
      COMMAND3
Label2:
      COMMAND1
      COMMAND2
```

```
Goto Label1
Label3:
```

As you can see, the GOTO command does not have to end the program. On the one hand, this is an advantage, because we work faster, but on the other hand, we need to remember about the specific behavior of the program. If you "jump" to a specific part, then the values of variables and the state of other elements used will not return to the original form. For example, if you increase the values of variables, your conditional statements can work differently. The line with the word GOTO is a very simple loop, to which you need to adjust the operation of the other parts of the program.

SUBROUTINES

The so-called subroutines are similar to labels. They also need to be marked with a specific name, but must be terminated with the word RETURN. Take a look at another example:

```
point2:
      COMMAND1
      COMMAND2
      COMMAND3
Return
```

It looks very similar to the last line. To call the subroutine, use the GOSUB command just as before. In our case it will look like this:

```
Gosub point2
```

There are no differences so far, because lines between the name of the subroutine and the word RETURN will be executed. However, later lines in the listing will not be called and the program will return to the place where the jump was made.

It causes that subroutines should be placed in one location of the program (usually at the end), which is independent of the other parts. This can be illustrated by the following structure:

```
Label1:
      COMMAND1
      COMMAND2
      COMMAND3
```

```
Label2:
      COMMAND1
      Gosub point2
      COMMAND2
End

point2:
      COMMAND1
      COMMAND2
      COMMAND3
Return
```

Now, after completing COMMAND1 belonging to "Label2", the program will run instructions belonging to the subroutine "point2", and then COMMAND2 will be called, still given inside "Label2".

Please note that we have made two more changes here. At the end of the program, but before the subroutine content, an END line was added - to terminate the program. If it was not there, the "point2" subroutine would be started, and then an error would be occured. This is because the RETURN command must "know" where it should return to. Calling a subprogram without a line with the word GOSUB is not valid, therefore the program must finish its operation earlier.

The subprogram was separated by an additional empty line from the rest of the listing. It is not necessary, but it makes the program more readable. In addition, you can use lines of so-called comments that will be skipped while the program is running. E.g:

```
; Here we go!
;
COMMAND1
COMMAND2

; Calculations
COMMAND3
```

Just use the semicolon character followed by any text in the comment. You can also enter the semicolon itself, it does not matter. This method is worth using in the complex programs, because after some time you may not remember how different parts work. In addition, the whole is easier to analyze and search, so we save time when testing and modifying the listing. You just have to remember that the program takes up a bit more space in memory.

PROCEDURES

Subroutines are convenient, unless we use a large number of different variables that are to influence the program many times. You must remember that variable names can not be repeated, moreover, in many cases you will have to enter further similar variables to make the individual parts of the program truly independent of each other.

All these problems will dissapear when you use the so-called "procedures", instead of subroutines. Let's say that this is a good way to place specific program functions in standalone modules. They can be called from the main part of the program, and variables are submitted independently of other values.

It should be emphasized that the content of the procedure must be in front of the line that will execute it. Otherwise, the program will not be able to run. Therefore, it is best to put all procedures in one part, at the beginning of the program - for example next to variable declarations.

The procedures have their own "workspace", so we can be sure that the other variables will not affect the operation of the functions. However, using procedures is more difficult. To create a new one you have to enter two lines with the words STATEMENT and END STATEMENT. The whole design looks like this:

```
Statement moja1{n}
    ...
    ...
End Statement
```

This time we need to use braces, and inside we give the parameters of the procedure. This is necessary, even if our procedure does not require any values.

Parameters are "arguments" that we put into the procedure as initial or input variables. You can use up to six variables to pass parameters to the procedure. If you need more, additional parameters can be placed in special global variables, which we discuss in the section under the heading "Local and global variables".

Of course, between the STATEMENT and END STATEMENT lines, we put commands to execute, just like in the subroutines. To later call the procedure, use the line with the name of the procedure as follows:

```
moja1{5}
```

Again, we use braces, but inside we already provide a specific value that will be used as a variable in the procedure. For example, to add a specific value to a variable and display it on the screen, you can use the procedure below. The last line means the calling the action:

```
Statement moja1{n}
      a=1
      a=a+n

      NPrint a
End Statement

moja1{5}
```

Thanks to the last line the variable "n" will get the value 5, which will be added to the initial value of the variable "a". The result of the calculation will be displayed using the NPRINT command.

If you call the procedure by entering a different number as a parameter, a different sum will be calculated according to the pattern inside the procedure. It can be much more complex, and the program will work differently after entering a line with one word FACT.

It should also be remembered that every variable within the procedure is declared again on every call, therefore the program may run a bit slower than using subroutines.

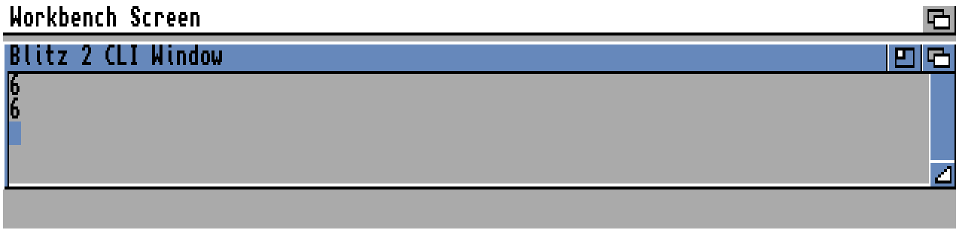
The procedures are independent of the other parts of the program. Variables retain their values only in their own "work areas". Thanks to this, each procedure can be copied to another program without any changes.

LOCAL AND GLOBAL VARIABLES

If necessary you can change the behavior of variables so that their contents can be read from the main program, i.e. outside the procedure. To do this, use the SHARED command inside the STATEMENT ... END STATEMENT construct. Referring to the previous example, it can look like this:

```
Statement moja1{n}
     Shared a
     a=1
     a=a+n

     NPrint a
End Statement

moja1{5}
NPrint a

End
```

Next to the word SHARED, simply give the name of the variable. Here is the result of the above program:



```
6
6
```

Please note that the first result is displayed from the procedure and the second - the main program. If you delete a line:

```
Shared a
```

then the second result of the NPRINT line will be "zero", because the variable will be treated as "local", i.e. readable within the procedure only. This is the difference between these two types of variables.

EXPANDING THE PROGRAM

INFORMATION FROM THE USER

Earlier, we were talking about displaying information, but in many cases you will want to make the program's activity dependent on information entered by the user. This is possible using the EDIT() and EDIT$() functions.

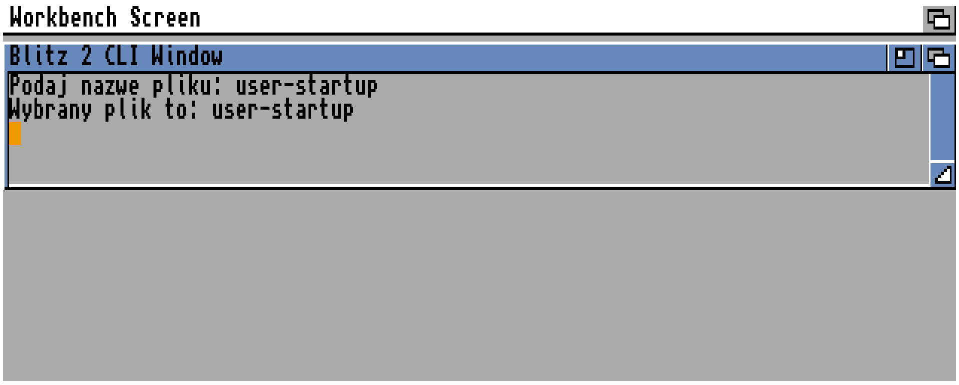
Use the first one when you want to get numerical data, the second - when the information you enter is to be text. We also write about the EDIT$() function in the "File Handling" section.

See an example:

```
Print "Enter file name: "
a$=Edit$(30)
NPrint "Selected file is: ",a$
MouseWait
```

This short program presents the use of the mentioned function. It should be assigned to a variable, according to the type of information entered. In our case, this is a text that is displayed using the usual NPRINT command. At the end, the program waits for the left mouse button and finish operation.

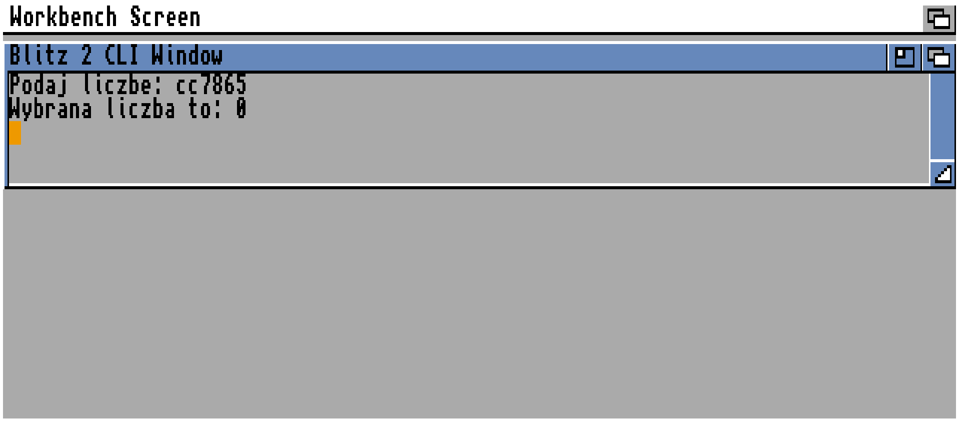
As an argument to both functions, you must enter a number indicating the maximum length that will be taken by the variable, that is "a$". Remember that this does not mean that you can not enter more characters, but the size you specify will be remembered as part of the variable. See the result of the above program:



```
Podaj nazwe pliku: user-startup
Wybrany plik to: user-startup
```

Note that if you use the EDIT$() function to save a string, you can easily include a number inside, even though you will have to replace it with a numeric variable later.

In contrast, using the EDIT() function to store a numeric variable causes the program to take an incorrect value or 0 (zero) after entering a character other than digits. For example:



```
Podaj liczbe: cc7865
Wybrana liczba to: 0
```

When you want to use a decimal fraction as a separator, enter the dot character, not the comma, i.e. for example:

```
22.537
```

Many other Basic dialects carry out a similar operation using a command (for example, INPUT), which require to give a variable without using a function. In Blitz Basic, the design of the program must be different, but the operation is very similar.

SCREENS

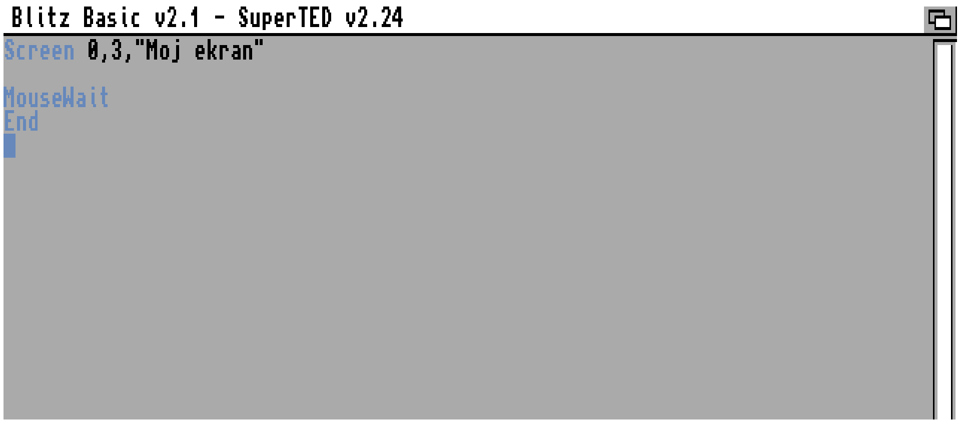
Blitz Basic enables direct support of the graphical interface elements of the operating system. So you can open a new screen just as almost every superior Amiga program does. You can use a number of commands, which mostly have the word "Screen" in their name.

- Opening the screen

If you want to open a separate screen for your program, use the short SCREEN command. All you have to do is give three basic parameters:

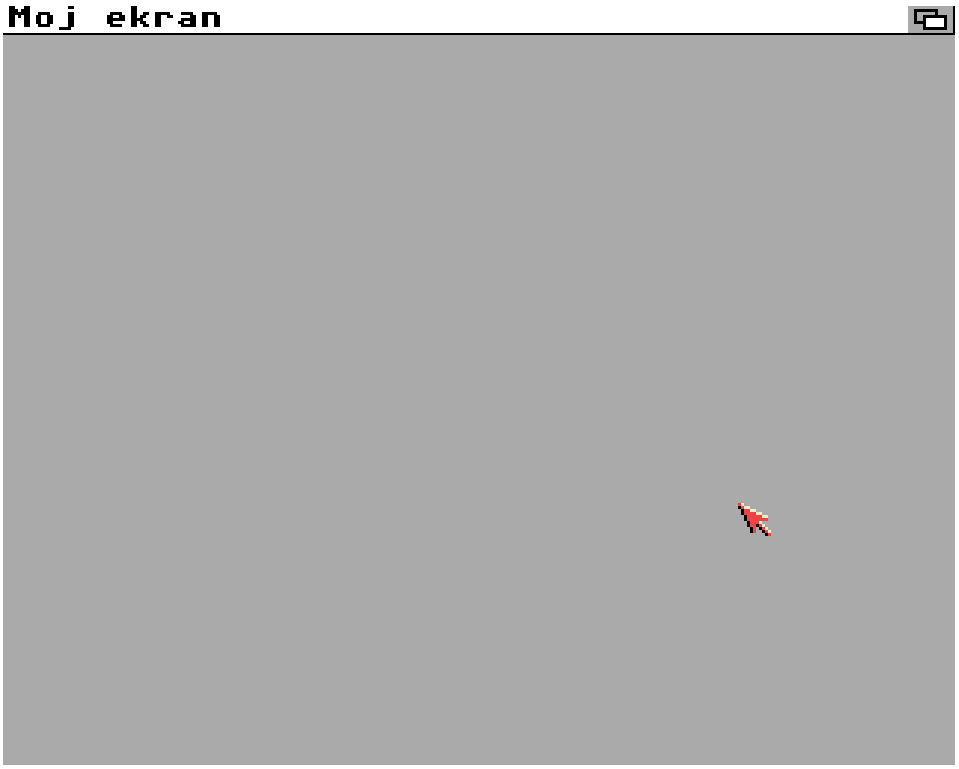
- screen number,
- display mode,
- screen title.

This may look like this:



```
Screen 0,3,"Moj ekran"

MouseWait
End
```

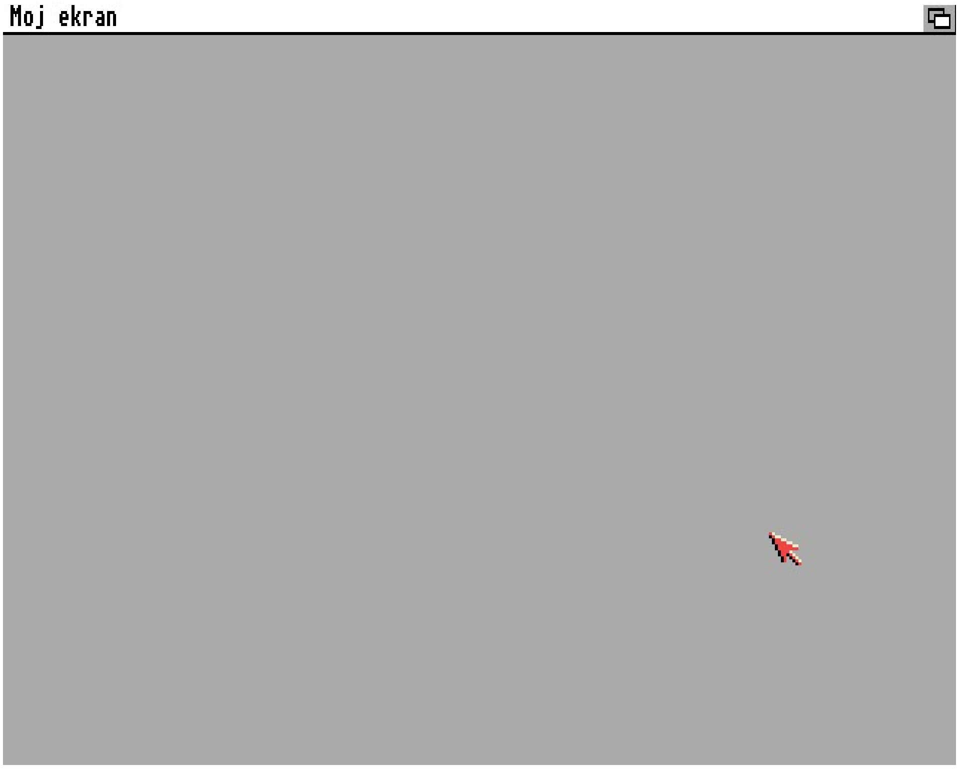
And here is the result of the operation of this simple program:



The last line means waiting for the left mouse button to be pressed. I write about it also in the separate section.

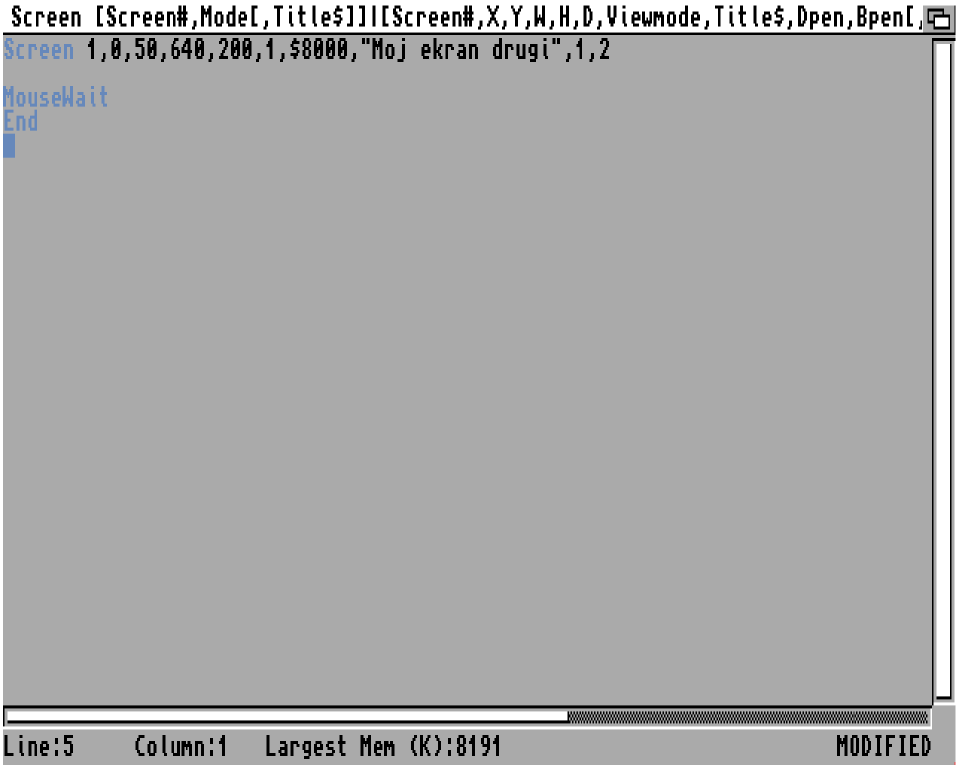
The parameter defining the display mode is the second and we enter it in the form of a number. It determines the so-called "depth" of the screen, i.e. the number of bitplans that will be available. Of course, this affects the available colors, for example 4 bitplanes have 16 colors. To calculate the quantity, the number 2 should be raised to the power indicating the number of bitplans.

Numbers from 1 to 6 are Lowres mode, i.e. a resolution of 320x256 points. If you need to use the Hires screen (640x256 pixels), add 8 to the mode number. The resolution will be changed, which can be seen especially on the titlebar. For comparison, see the next illustration:



It is also possible to run the program on the screen in Interlace mode, but in this case, we should add the number 16.

You can also use the SCREEN command in a more sophisticated way. However, you must enter 10 arguments in a different order. Take a look at an example:

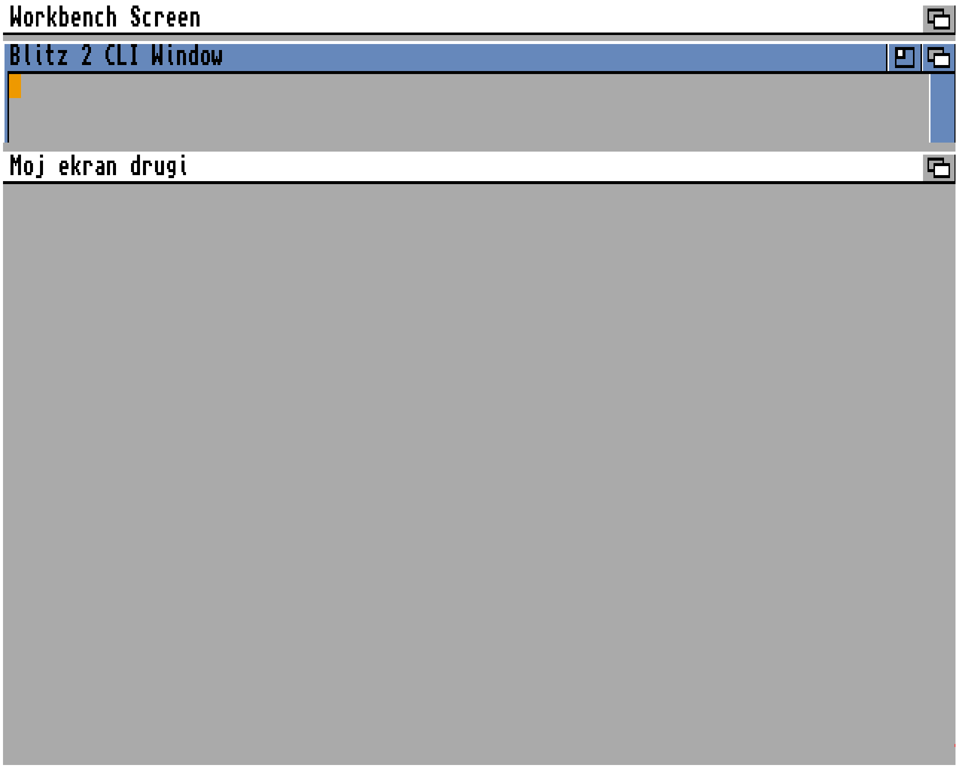


```
Screen [Screen#,Mode[,Title$]]|[Screen#,X,Y,W,H,D,Viewmode,Title$,Dpen,Bpen[,
Screen 1,0,50,640,200,1,$8000,"Moj ekran drugi",1,2

MouseWait
End
```

```
Line:5     Column:1    Largest Mem (K):8191                    MODIFIED
```

Using the visible line, you will open a screen with a resolution of 640x256 with a minimum of 2 available colors (1 bitplane). This is because we used the argument written as the number $8000.

Please note that we not only have more data in the line, but also have different types. Here is the effect of the program:



As you can see the screen is automatically moved vertically, so the upper part of the Workbench with the "CLI" window is still visible. This is all possible, because the SCREEN command requires entering arguments that significantly affect the features of the screen. These are the following elements, in order:

- screen number,
- horizontal position of the screen,
- vertical position of the screen,
- horizontal resolution,

- vertical resolution,
- the "depth" of the screen,
- video mode,
- screen title.

The last two numbers indicate the colors that will be used - we'll talk about them in detail later.

Other items, besides video mode, should be understandable to you. Here you must find specific values meaning:

- 0	- Lowres mode,
- $8000	- Hires mode,
- 4	- Interlace mode (512 pixels vertically).

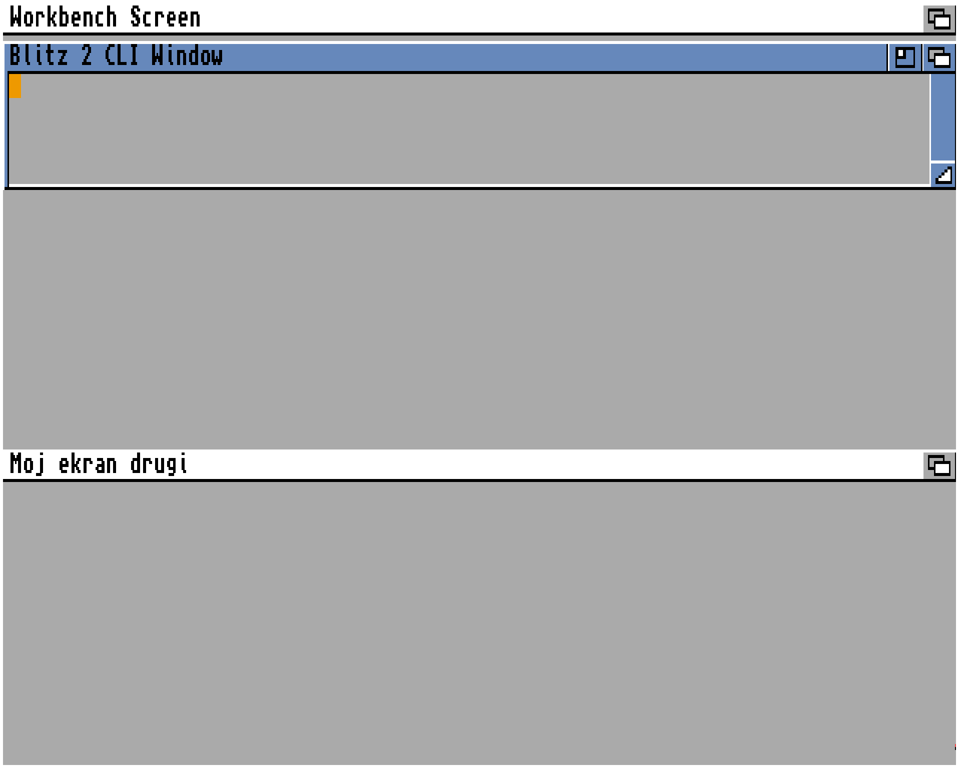
- Changing the position of the screen

Once we have the screen ready, we can do a series of further steps on it. A characteristic feature of the Amiga system is the ability to drag the screens. Usually, we do it with the mouse when we launch a larger number of programs. However, nothing prevents you from using it in your program.

The screen can change position according to the numbers given in the line containing command named MOVESCREEN. All you need to do is enter the distance from the upper left corner where the screen is to be moved. E.g:

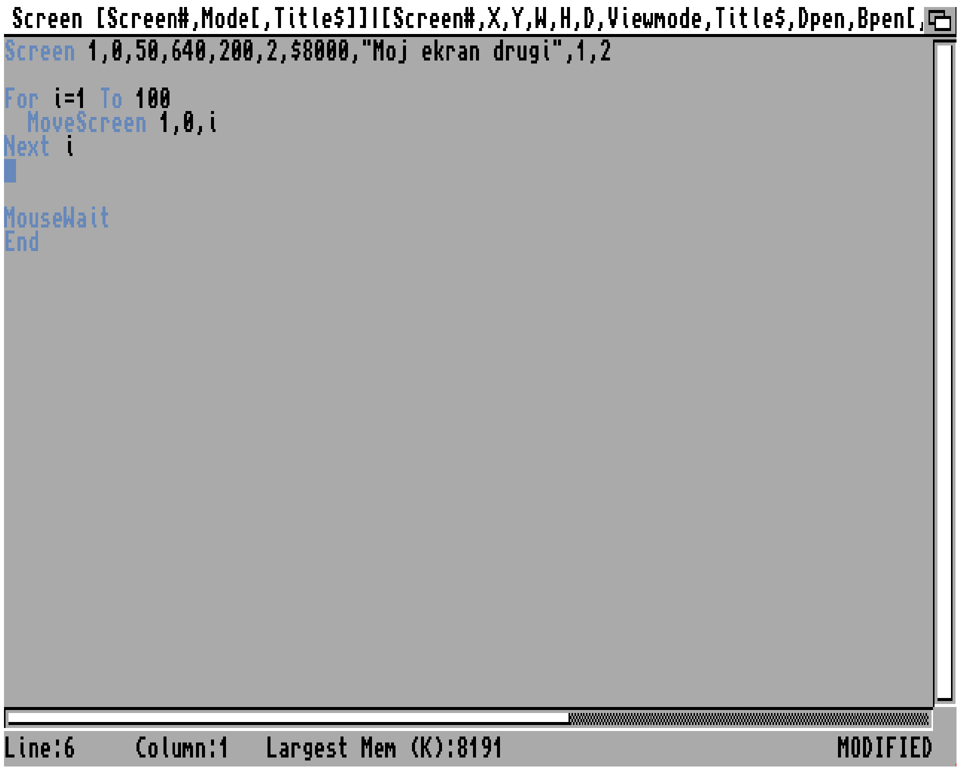
```
MoveScreen 0,0,100
```

See what will be the result of the program, which first - opens a new screen, and then - moves it with the values given above:



Of course, the user does not observe the moment of changing the position of the screen, everything happens so fast that you just see the screen in another position.

With this function you can smoothly move the screen, just create a loop that modifies the number next to the word MOVESCREEN. Here's another example, this time a part of the program visible in the "Ted" editor:


```
Screen [Screen#,Mode[,Title$]]|[Screen#,X,Y,W,H,D,Viewmode,Title$,Dpen,Bpen[,
Screen 1,0,50,640,200,2,$8000,"Moj ekran drugi",1,2

For i=1 To 100
  MoveScreen 1,0,i
Next i


MouseWait
End
```

```
Line:6     Column:1    Largest Mem (K):8191                    MODIFIED
```

This time the effect will be visible. In utility program, it is difficult to apply such an action, but you can easily use it when writing a game.

Please note that you can open several screens and change their positions relative to each other. We have such effects in many games for Amiga. However, to effectively manage multiple screens, you should know the commands that allow you to change the order, temporarily hide and display relevant fragments when necessary.

The simplest operation is to hide the screen with a specific number. To do this, enter a line similar to the following:

```
HideScreen 1
```

Next to the command, we only give the number of the screen that was previously opened. It's just as possible to show the screen again, but we must use the word SHOWSCREEN, which means:

```
ShowScreen 1
```

Remember that if you create many different screens, their order may be different. An open screen does not have to be the one that is "on top." The command performs the additional operation that sets the screen with the given number as the first.

As a result, after you use the words HIDESCREEN and SHOWSCREEN, you change the order of the workspaces. If you write a program that works on several screens, and you place important controls on each of them, it can have a huge significance for the readability of the user interface thus created.

You may also need the option to find the screen with a specific number. The FINDSCREEN command is used for this, and you can additionally give the title of the screen.

Please note that after performing this function, the given screen will be automatically used.

This means that you can perform various operations on it without having to show results of your program to the user. In many cases, you will have a need to "quietly" change the screen content, so keep in mind this possibility.

You can also perform operations directly on the Workbench screen. You can do this by entering a zero number next to the FINDSCREEN command, so simply:

```
FindScreen 0
```

You can also assign your own number and then use it like other open screens. To make this possible, you must use the word WBTOSCREEN instead of FINDSCREEN.

After assigning a number to the Workbench screen it is possible, for example, to open own windows on it. In other words, your program can run on the desktop, but it does not exclude simultaneous operations on other screens. But remember that Amiga has limited graphics memory (Chip RAM). The more screens in high resolution and colors, the more you use the available area. Therefore, the WBTOSCREEN command can be useful also if you want to reduce the memory demand.

You can close the screen if necessary and it does not require any special skills. You just need to enter the CLOSESCREEN command and type the screen number. Before you do that, it's worth to learn about the other screen modification possibilities.

COLOR PALETTE

Each screen can have its own color palette. Colors in the Amiga operating system are represented as values of three basic components:

- Red,
- Green,
- Blue.

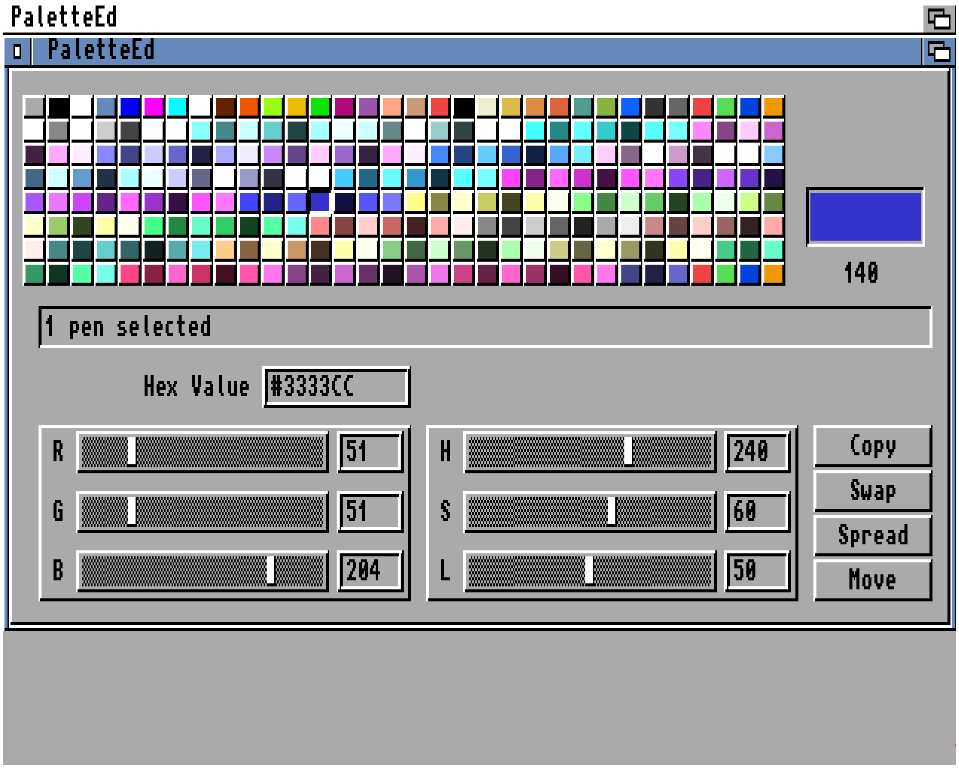
These values are saved in the form of the so-called RGB model. As you can easily guess, its name came from the first English letters describing the names of individual components.

Thanks to different combinations, we can receive a very wide range of colors. All the time you have to remember that the screen has strictly set number of available colors, but you can change their values.

Color sets are called palettes. In various programs for Amiga, you can modify the palette and save it in a separate file. An example can be the "PaletteEd" program found in the Aminet website, ie on the following page:

`aminet.net`

The next illustration shows the appearance of the color palette edit window:



I show it for a specific reason, because in Blitz Basic you can perform many operations on color palettes. You can load the palette information from an IFF file or define it with a few commands. The simplest word is RGB, which allows you to modify one of the palette colors of the screen which you use. Just type a line similar to the following:

```
RGB 1,0,15,15
```

Here we give the color number and component values always according to the same order - Red, Green, Blue. Each number must be between 0 and 15. This allows you to adjust the colors on the screen to your needs. Please note that changes are made immediately in this case.

Please note that different objects on the screen can use separate palettes. Therefore, in addition to determining the colors for the current screen, you can also change the set of colors assigned a specific number. We will achieve this using the PALRGB command, which should be used very similarly to the previous one.

The main difference is that at first you have to enter the pallete number. Here is an example:

```
PalRGB 0,1,0,15,15
```

The above two functions operate within the capabilities of Amiga graphics chips, but do not support the AGA chipset. As we know, it allows you to get more colors, so you should be able to set the colors more accurately. And as it really is, because there are equivalents of both commands intended for users of Amiga 1200 and 4000. They have the following names:

- AGARGB,
- AGAPALRGB.

The method of operation is identical, but now each component can take the value from 0 to up to 255. Remember that this difference is related to the construction of Amiga AGA chipset, so you will not get a similar effect on the Amiga 600 equipped with ECS chipset.

Thanks to the above functions, you can change specific colors in the palette, and more precisely, define their new values. However, the colors are not automatically modified on the screen. You must call the effect manually using the SHOWPALETTE command. Here, we only give the number, i.e. for example:

```
ShowPalette 2
```

This will change the colors in palette with number 2, which can be assigned to different screen elements.

Manual adjustment of all colors is not very convenient, that's why Blitz Basic gives you the opportunity to perform operations to save and load another palette from the disk. This requires the use of the SAVEPALETTE or LOADPALETTE command. To make this easier to understand, we will start with the option of saving. By entering a line like the following example:

```
SavePalette 0,″Work:file3″
```

you will create a file in the IFF format on the "Work" disk. This is important because such files are often associated only with graphics that can be loaded into programs such as "Deluxe Paint" or "Personal Paint".

However, Interchange File Format (abbreviated as IFF) was developed by Electronic Arts company in the mid-1980s to be used in a much wider range. It can store different types of data, including graphics and sound.

The Amiga system is very much connected with the IFF file format and our programming language behaves in the same way. Therefore, the color palette will be saved in the IFF CMAP file. From a practical point of view, you only need to know that such a file does not contain information about graphics, but only the definition of a set of colors.

The loading of the palette is not substantially different from the save operation. You can use such a line:

```
LoadPalette 1,"Work:file5"
```

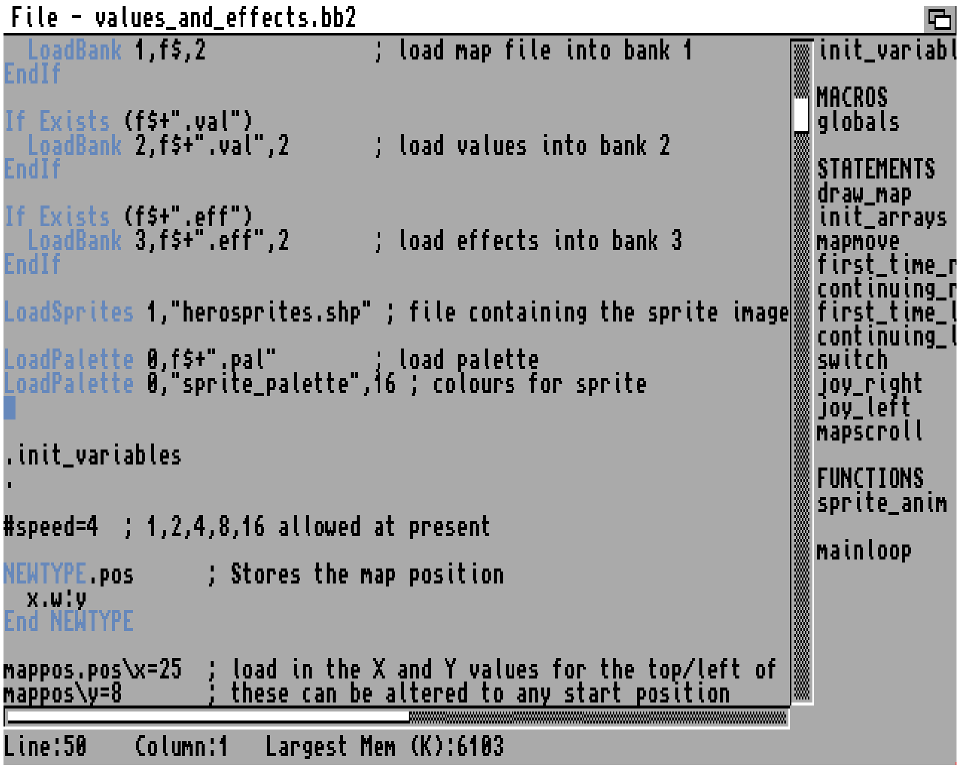
to load a palette from a previously saved file. It is also possible to "extract" a palette from an IFF ILBM type file, i.e. a graphic that can be displayed on the screen. In this case, the program will only read the information about the color set and will not cause any other changes on the disk. Thanks to this, you can use various IFF files without worrying about data corruption.

Separate color palettes can also be prepared in other programs. An example can be the already mentioned "Personal Paint" graphic editor.

You can manage palettes in many programs, but you must remember to always save files in the IFF format. Otherwise you will not be able to use them later by writing a program in Blitz Basic, unless you use much more complex operations. Therefore, it is better not to change the default format, at least at the beginning.

Loading the palette with the LOADPALETTE command - just like before - does not automatically change the screen's appearance. To make changes, use the USEPALETTE command by entering the number of the palette.

For ease of use, see how it can look in a more expanded listing:



File - values_and_effects.bb2

```
  LoadBank 1,f$,2              ; load map file into bank 1
EndIf

If Exists (f$+".val")
  LoadBank 2,f$+".val",2       ; load values into bank 2
EndIf

If Exists (f$+".eff")
  LoadBank 3,f$+".eff",2       ; load effects into bank 3
EndIf

LoadSprites 1,"herosprites.shp" ; file containing the sprite image

LoadPalette 0,f$+".pal"         ; load palette
LoadPalette 0,"sprite_palette",16 ; colours for sprite


.init_variables
'

#speed=4  ; 1,2,4,8,16 allowed at present

NEWTYPE.pos       ; Stores the map position
  x.w:y
End NEWTYPE

mappos.pos\x=25  ; load in the X and Y values for the top/left of
mappos\y=8       ; these can be altered to any start position
```

```
init_variabl

MACROS
globals

STATEMENTS
draw_map
init_arrays
mapmove
first_time_r
continuing_r
first_time_l
continuing_l
switch
joy_right
joy_left
mapscroll

FUNCTIONS
sprite_anim

mainloop
```

Line:50   Column:1   Largest Mem (K):6103

You can also copy a palette to use it in another way, such as a different screen. To achieve this, use the DUPLICATEPALETTE command as follows:

```
DuplicatePalette 1,2
```

As arguments we give the number of the source palette and then the target one. This word has no additional features, but it can be useful, for example, when you want to display a graphic created in the same graphics program or treat one of the palettes as "working data".

In addition to manipulating the palette, it is also a useful option to read information about specific colors. You will not always want to change them, sometimes to achieve the desired result, you should just copy the color and use it in relation to another screen or other element.

That's why Blitz Basic provides commands that you can use to read information about used colors. They have names convergent with components in English, that is:

- RED,
- GREEN,
- BLUE.

Just give them the color number you want to check, for example:

```
a=Red(1)
```

As a result, you get the value of the color component, which will correspond to the settings possible to perform using the RGB and PALRGB commands which have already been discussed. So they will be numbers from 0 to 15.

Earlier, I wrote that there are separate functions for users of AGA chipset allowing for greater accuracy - from 0 to 255 for each component. Similarly, you can use commands beginning with the "AGA" symbol, which have an analogous function.

These are the words as below:

- AGARED,
- AGAGREEN,
- AGABLUE.

PULL-DOWN MENU

In addition to using typical screens, your program may have its own pull-down menu. This is nothing special, because almost every major program contains the menu. Before we present commands to create a menu, it is worth adding how to plan its construction.

In theory, the menu can contain any elements, but note that most programmers use the scheme known from the operating system. It unifies the support of standard functions, and for the user it means easier use of your program. Let's take a look at the basic assumptions.

The pull-down menu is element characteristic of the Amiga system. It is displayed at the top of the screen where its titlebar is located. Unlike other operating systems, the menu is not visible all the time while you work. It should be called with mouse.

To do this, move the pointer to the screen titlebar, then press and hold the right mouse button. The names of individual menu groups are shown on the top. To display the options of each group, move the pointer to the name, while holding the right mouse button.

All options are displayed in the form of a list, according to the movement of the mouse pointer. If we hover over the option name, it will be highlighted, but not yet launched. Only after releasing the right mouse button, the option will be called and the entire menu will automatically disappear.

The pull-down menu is usually associated with the screen. So regardless of what part of the program we use, we will always get the same options. However, it can also be associated with a specific window. Then, depending on which window is active, other menu functions will be displayed on the screen titlebar. This situation is particularly important when we run many programs on the same screen, so we will open several windows.

If we use a program that opens its own screen, we immediately access its menu. But if the program opens its window, we must activate it, otherwise its pull-down menu will be unavailable. It is rare that we will not have any options to choose, most often we will get access to one of the menus, not necessarily the one we are interested in. Therefore, when working with many programs at once, you should pay special attention to the screen you operate on and which window is active.

Note also that not all menu options are active. Some names may have a darker color and you can not choose them. This means that at the moment they are unavailable and will be activated automatically when it is possible, eg when the relevant section in the program is launched.

Some menu options may also have the so-called sub-menu. In this situation, moving the mouse pointer over the name will display a list with additional options. This happens when a given function provides more detailed possibilities and we have a large number of them. Placing every feature in the main menu is then inadvisable or even impossible due to lack of space on the screen.

The operation of the sub-menu is analogous to the regular menu. Choosing a specific option usually results in the program launching, displaying a message or opening an a screen window. There are also options whose selection only changes the status of the function, i.e. "enable" or "disable". It is easy to get to know it, because once selected, it will be marked with a characteristic sign on the left. Turning on the option may result in additional program activities, but this is not the rule.

- Typical menu structure

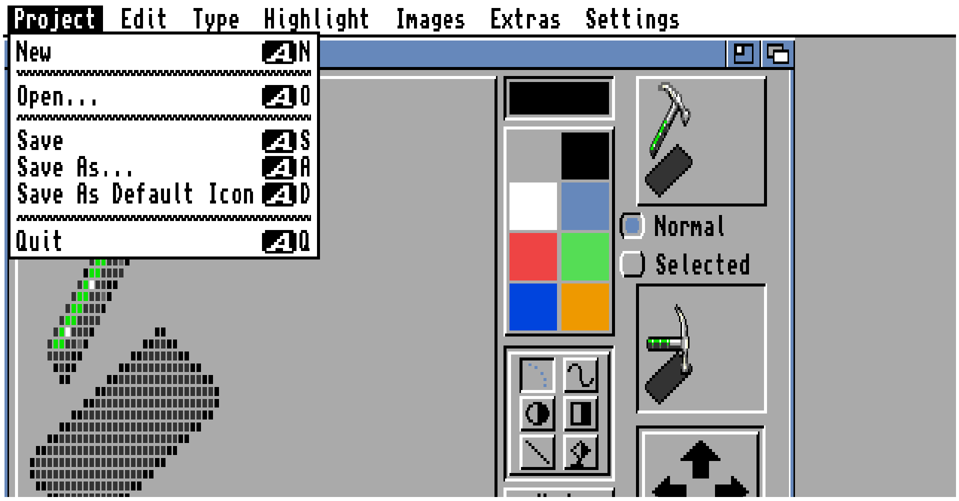
In most programs, we see a lot of options that you can also found in the basic system programs. Note the typical menu layout. It contains items such as:

- "Project",
- "Edit",
- "Settings" or "Preferences".

The "Project" menu is always related to the product of the program's operation, i.e. the file saved on the disk. Here you can find the functions of file operations:

- "Open" or "Load",
- "Save" and "Save As",

In addition, we have a feature of creating a new project - "New", as well as basic information about the program and the function to exit the program - "Quit".



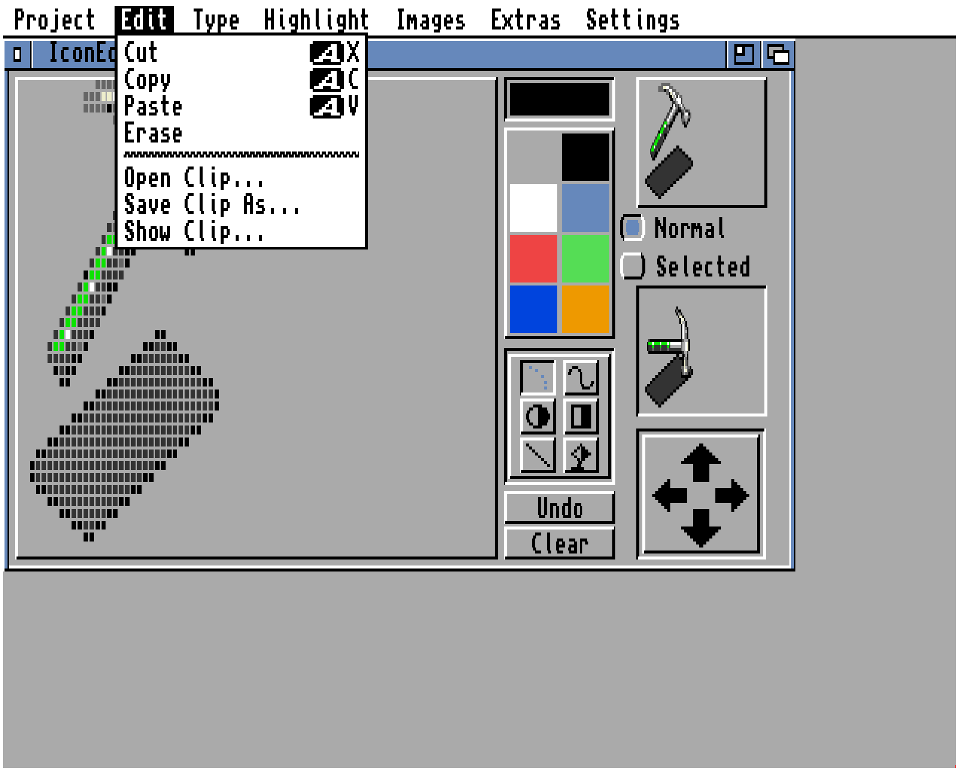
Thanks to this, the most typical functions are identical and it is easier to learn how to use. This standardization progressed with the development of the operating system, so the newer the program, the greater the chance that we will find typical elements in it. Remember that when you write your own program, you should not create your own solutions, but rather use the listed elements, unless it is absolutely necessary to change them.

It is also worth noting how is the difference between the "Save" and "Save As" options. If the user saves the project for the first time, both work the same. After selecting them, a selection window appears on the screen, where we indicate the name of the file with our project. However, if the project has already been saved, the "Save" function automatically saves it under the same name, losing the previous data. If you want to save the file under a different name, use the "Save As" option. It always displays a selection window where we can decide on the file name.

The second typical menu is "Edit". It contains options related to operations on individual project elements. The most common functions are as follows:

- "Cut",
- "Copy",
- "Paste".

These options use a system Clipboard. Its wider use in the Blitz Basic program will be discussed later.



The "Cut" option should enable - according to its name - to cut out the previously selected part of the project and save it temporarily in the computer's memory. The method of selecting depends on the specific program, usually consists in highlighting a fragment of the content using the mouse.

The selected element is saved and can be later inserted in another place using the second "Paste" option. You should also foresee the possibility of remembering the selected part without cutting it so that you can duplicate the content of the project. The "Copy" option is used for this.

The last standard menu is an item called "Settings" or "Preferences". It is located almost always on the right side of the screen, as the last or penultimate menu. It has options to load or save program settings - respectively using the function:

- "Load Settings",
- "Save Settings".

You can also find the option "Save Settings As", which has the same function as "Save As", but of course concerns the program settings file, not the project.

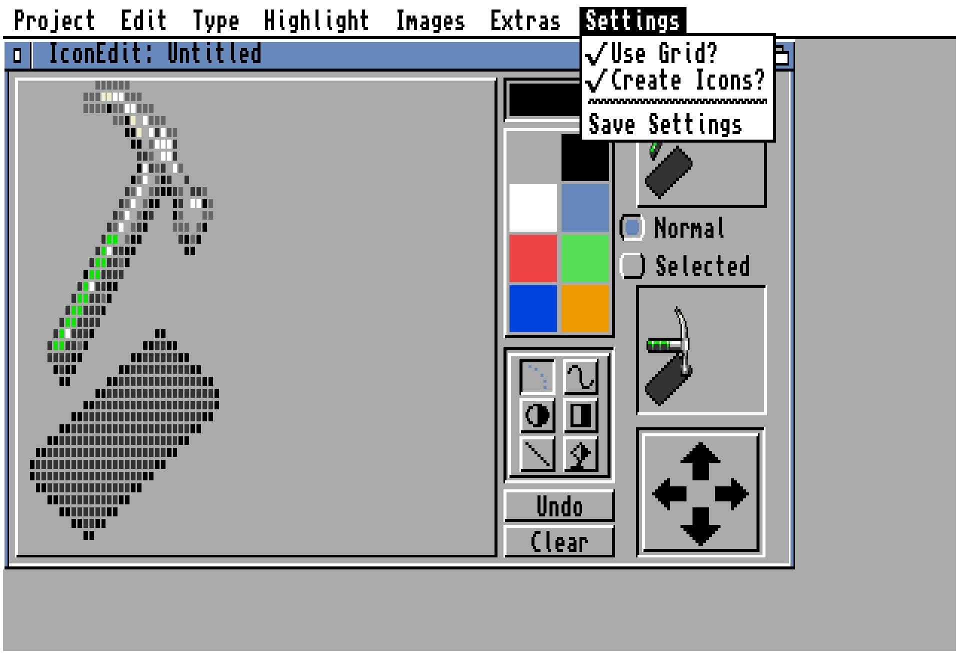
The function "Save Icons?" is also very often inserted into the menu. If selected, icon files will be saved along with files containing settings. Thanks to this, it is possible to load the settings by pointing the icon with the mouse on the Workbench, although it will not be possible in every case.

If your program allows you to save your own project files, you should also add automatic saving of icons. Remember that the user should get the support so that he can operate the program in the simplest way.

Some programs also have options with the following names:

- "Reset To Defaults" or "Default Settings",
- "Last Saved".

They are used for automatic loading of settings files, but according to strictly defined rules. The first function evokes the default program values created by author. The second loads the settings file saved by the user.



I have not written anything yet about the purely informative menu option which is the "About" function. It should display a small window with basic information about the program, its version, author and release date. You can also place here instructions for contacting you or for obtaining detailed program documentation.

If you provide various editions of the same program, for example you would like to use the registration that increases the number of available functions, you should also give the edition symbol, the name of the person registering or the license number in the "About" window. Some also write here donation requests, but this is not the best place for such messages.

All these comments should be taken into account when creating the pull-down menu in the program. Of course, you do not have to use all the elements, because everything depends on the purpose of your finished product.

It is best to act in accordance with the principle that you should add functions to the menu on a regular basis, when writing subsequent program modules, and do not create names of the items that do not work. Nevertheless, I think that the basic elements, such as the "Project" or "Edit" menu, should be planned in advance, and later you should only expand the menu depending on the needs.

- Creating the menu

Once you know technically how the menu should look, it's time to apply this information in practice. To create a pull-down menu, use a group of commands with names beginning with the word MENU.

The first element is MENUTITLE that will be used to add a menu title, and later will be contained various options. Next to the command you have to provide:

- the number assigned to the entire menu we create,
- menu title number,
- the content of the title.

This may look like this:

```
MenuTitle 0,0,"Project"
```

Thanks to the above line you will create the first menu called "Project". Note that the ordinal number is counted from zero. You have to add functions to the "title", for example "Open" or "Save". This is done using the MENUITEM command. It allows you to add new item to the element created by MENUTITLE. Of course, the options will appear vertically, below the title.

In this case, you should enter the following information as arguments:

- menu number,
- so-called "flags",
- menu title number,
- option number,
- option name,
- keyboard shortcut calling the option.

Here are examples of valid entries:

```
MenuItem 0,0,0,0,"Open... ","O"
MenuItem 0,0,0,1,"Save ","S"
MenuItem 0,0,0,2,"About... ","A"
```

The menu number must be the same as the one we first entered in the MENUTITLE line. Otherwise the line will refer to another pull-down menu, and the screen should have only one group of functions called with the right mouse button. The title number of the menu should also be the same as before, in order to add the item to one group.

Next in order are: the number and name of the option. The functions should have consecutive numbers. The name can be any text, but you should take into account the comments about the menu structure and clearly indicate the function being called.

In this case, the keyboard shortcut means only one character, which will have to be used together with the AMIGA key to alternatively activate the option assigned to the given item.

Here you should use the experience of other programmers, so check what keys usually call various operations. However, this is less important, because your program can have its own shortcuts to other functions.

Besides, everyone will be able to check the shortcuts, because the symbol will appear next to the option name, just like in other programs.

- Adding the options

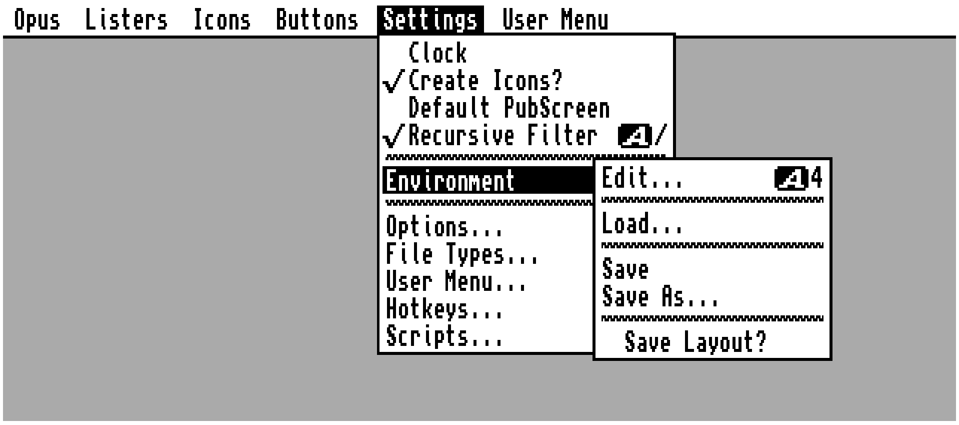
We have not said anything yet about the very important feature which is the "flags", written as the second in the line containing MENUITEM command.

Let's remind how it looks like:

```
MenuItem 0,0,0,1,"Save ","Z"
```

A flag is a specified number indicating the type of option being added. In the above line, we have zero, so it will be the usual position that the user chooses to activate the function. Of course, "Save" should immediately open a selection window, which is why this type fits best.

You certainly know that options can take other forms. For simplicity look at the illustration:



If you want to get a similar effect, you need to change the value of the "flag". And so, the number 1 causes that the option will be possible to be "turned on" and "turned off", just like in the window on the Workbench screen. Value 2 creates a special menu item that does not belong to standard elements.

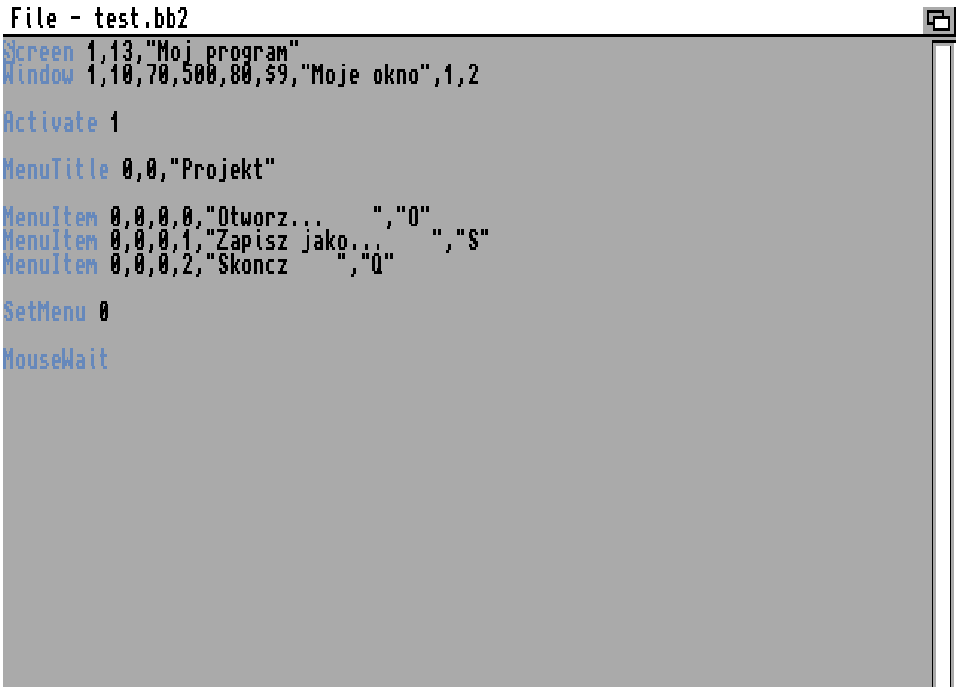
All items that appear in the same menu will be mutually exclusive. This means that only one of them can be "active" at once. If this happens, all other menu items will be automatically "turned off".

This last feature can be very useful wherever you want to give the user a narrow selection of options. It is true that such a menu is rare, but it does not break the operating system standards, so you can easily use it. But you need to rebuild the functions so that only one option can be active at any one time.

The entire menu structure should be prepared according to the above scheme. The number of titles and items is arbitrary, but the options should be logically grouped. In addition, a single list can not be too long, because it will not fit on the screen.

It's best to assume that the user uses the standard resolution of 640x256 points and the "Topaz" font in size 8 pixels. Use these parameters as a starting point. If you must necessarily use Interlace mode or other typefaces, do not hesitate to do it, but it should not be the only possible program setting. It's always better to split the options on several menu titles than to put everything under one list.

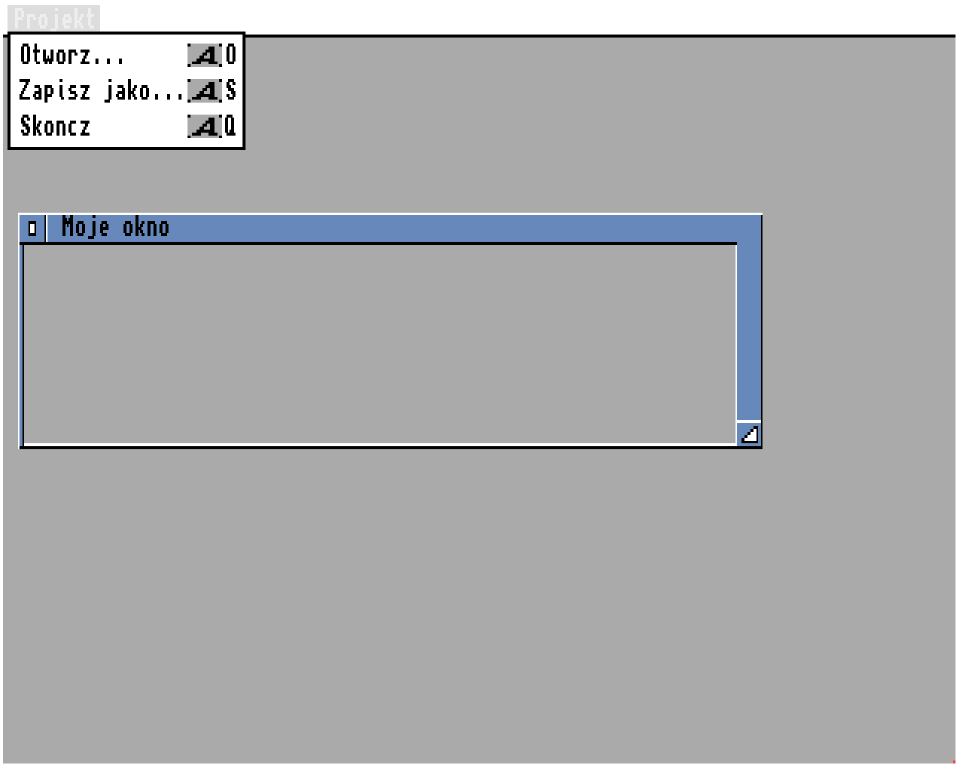
See a ready example of how to create a pull-down menu. At first, the listing:



File - test.bb2

```
Screen 1,13,"Moj program"
Window 1,10,70,500,80,$9,"Moje okno",1,2

Activate 1

MenuTitle 0,0,"Projekt"

MenuItem 0,0,0,0,"Otworz...    ","O"
MenuItem 0,0,0,1,"Zapisz jako...    ","S"
MenuItem 0,0,0,2,"Skoncz     ","Q"

SetMenu 0

MouseWait
```

And now, after launching the program:



As you can see, in addition to the lines that make up the menu, we have several other elements that we will discuss in a moment. The program opens the screen, the active window, and then assigns it pull-down menu called "Project".

You can assign the menu to the active window using the SETMENU command. We give it only the number assigned to the entire menu we create, i.e. the same first item as in the previous functions. Of course, one window can have only one menu set.

If you want to get a more unusual appearance of your program or use a specific color palette, you can cause the menu text to be displayed with another color. Enter only a line similar to the following:

```
MenuColor 2
```

You only need to enter the color number here. Please note that the above line should be called before the commands creating the menu.

- Menu activation

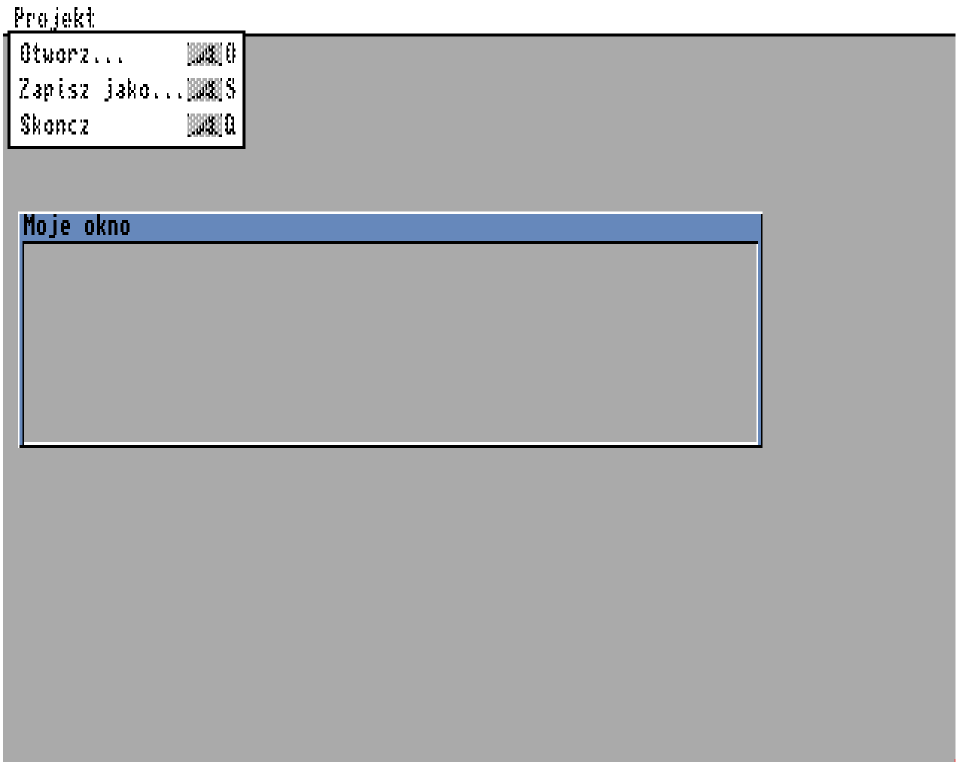
The menu can be turned on and turned off if necessary. The MENUSTATE command is used for this. You can choose whether you want to deactivate the entire menu or just one part. To be able to do so, along with the name of the command you must provide:

- number assigned to menu.
- menu title number,
- option number,
- word ON or OFF.

The first two items coincide with previous commands and do not require a comment. The option number should also not be unclear, although in this case an important note must be added. If the option contains a "sub-menu", this way you enable or disable the entire section.

Additionally, before the last argument, you can enter the "sub-item" number, if you want to turn off only one of them. Remember that the menu and options will be visible all the time, but some features will be inactive. This behavior can be observed in any program containing the pull-down menu as well as on Workbench.

See how the menu from the previous picture will look like when we turn it off entirely:



This effect will be obtained after typing a shortened version of the MENUSTATE command, where you will only indicate the menu number, as below:

```
MenuState 0,Off
```

This is the easiest way to block all functions when you do not want the user to cause a program error, for example when performing disk operations or other labor-intensive tasks.

- Checking the status of the option

If the finished menu contains items that can be activated, you will need to check the status of these functions. The user can change them at any time and the program must respond appropriately to modifications. Of course, a specific action depends on your ideas, but you need to know how to check the status of the options.

Blitz Basic provides a function called MENUCHECKED() in this area. You have to enter the same information as during "turning on" and "turning off" the item, i.e.:

- menu number,
- menu title number,
- option number,
- "sub-item" number (optional).

The whole thing may look like this:

```
a=MenuChecked(1,0,0)
Nprint a
```

The function in result will give typical values, that is 0 (zero) or -1. The first means that the option checked is inactive, the second indicates "enabling" the function.

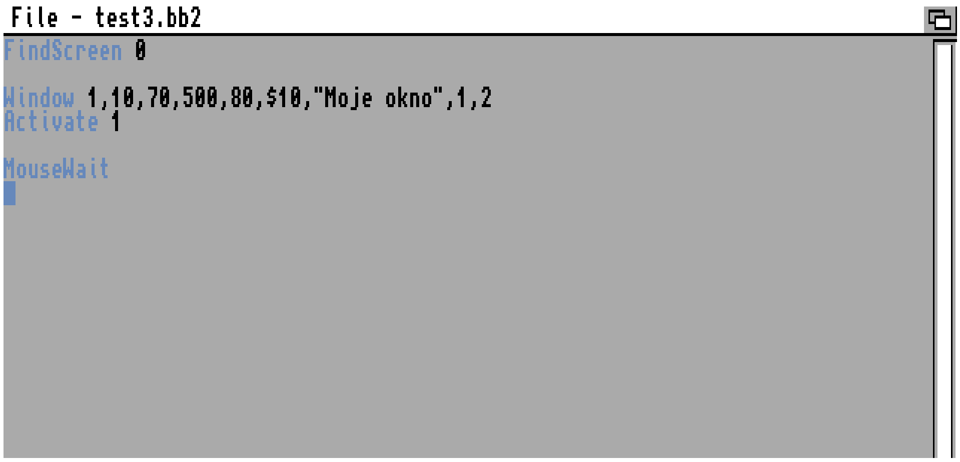
Of course, these are logical values meaning FALSE or TRUE. You can read more about this topic in the section entitled "Operators".

WINDOWS

Windows are a basic element of the graphical interface in the Amiga system. Almost every program uses them, and even if you run only the AmigaDOS command, you enter them in the "Shell" window. If you want to create a new game, you may not want to use standard windows, whereas in the case of a utility program you should keep all possible elements of the operating system.

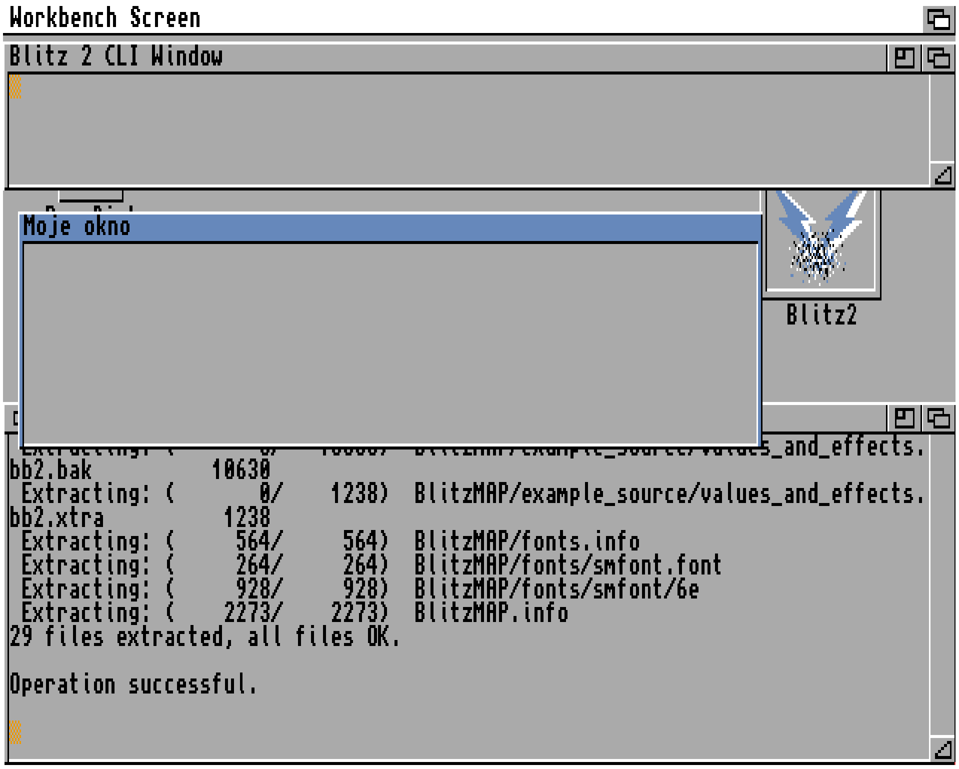
Blitz Basic allows you to use windows, just like screens. There are a number of commands at your disposal that you can use to open, close windows, and to call many of the operations you need when operating your program.

First, look at the example program. This is a modified version of the listing from the previous section, which opens the window on the Workbench screen:



```
File - test3.bb2
FindScreen 0

Window 1,10,70,500,80,$10,"Moje okno",1,2
Activate 1

MouseWait
```

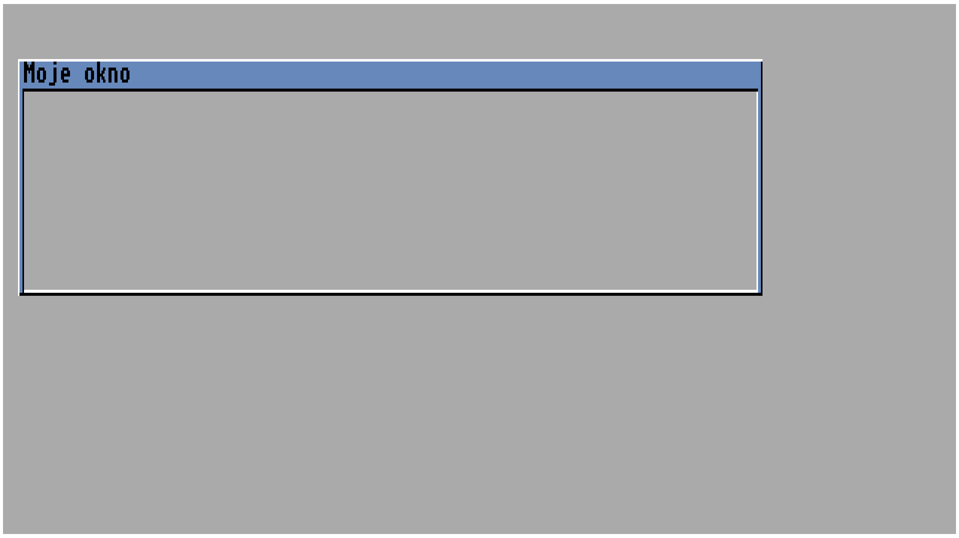
And here is the result of its operation:



```
 Extracting: (        0/    10630)  BlitzMAP/example_source/values_and_effects.
bb2.bak           10630
 Extracting: (        0/     1238)  BlitzMAP/example_source/values_and_effects.
bb2.xtra           1238
 Extracting: (      564/      564)  BlitzMAP/fonts.info
 Extracting: (      264/      264)  BlitzMAP/fonts/smfont.font
 Extracting: (      928/      928)  BlitzMAP/fonts/smfont/6e
 Extracting: (     2273/     2273)  BlitzMAP.info
29 files extracted, all files OK.

Operation successful.
```

You can create such a window in a very easy way. All you need to do is use the WINDOW command. Remember that the window will be displayed on the active screen, so you have to open your own - as we discussed earlier - or use the Workbench screen.

The standard window has buttons on the frame, but you've certainly seen windows without some elements. See how the same window looks open on a separate, blank screen:



Therefore, the word WINDOW also has several arguments by which we set window parameters. To open the window in the simplest way, you must enter the following arguments along with the command name:

- window number,
- horizontal coordinate of the upper left corner,
- vertical coordinate of the upper left corner,
- window width,
- window height,
- so-called "flags" assigned to the window,
- window title.

This means using line according to the following formula:

```
Window 1,100,100,300,50,$9,"My window",1,2
```

Most of the items do not require a comment, but what are the mysterious "flags"? These are the values that define window parameters, for example, whether it should contain a close button, it is possible to change the size or other. The "flag" value should be given starting with the dollar sign ($). Here is list of the "flags" with a brief explanation:

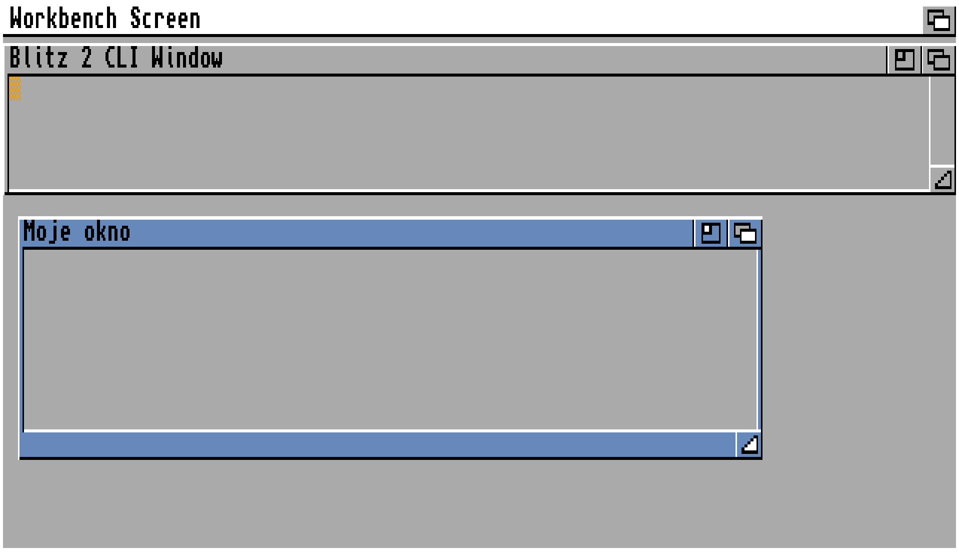
- **$0001**	- adds a resize button in the bottom right corner of the window,
- **$0002**	- allows you to change the position of the window with the mouse,
- **$0004**	- the window will be able to be located "in front" or "behind" other windows,
- **$0008**	- adds a window closing button in the upper left corner,
- **$1000**	activates the open window,
- **$0800**	- causes the window to have no frames,
- **$0100**	- the window will be open "behind" all other currently available windows.

- **`$0400`**	- causes that the elements placed inside the window will not be able to cover the frame, it is convenient, but it takes up more memory.

After determining which items you want to use, the values should be added up, and the resulting value should be entered as one argument. For simplicity, see another example:

```
Window 1,0,0,200,35,$25,"My second window",1,2
```

In the above line we have the value "$25", which we received from the summation of the first five items from the "flags" list. The window will now have a resize button and standard fields in the upper right corner. It looks like this:

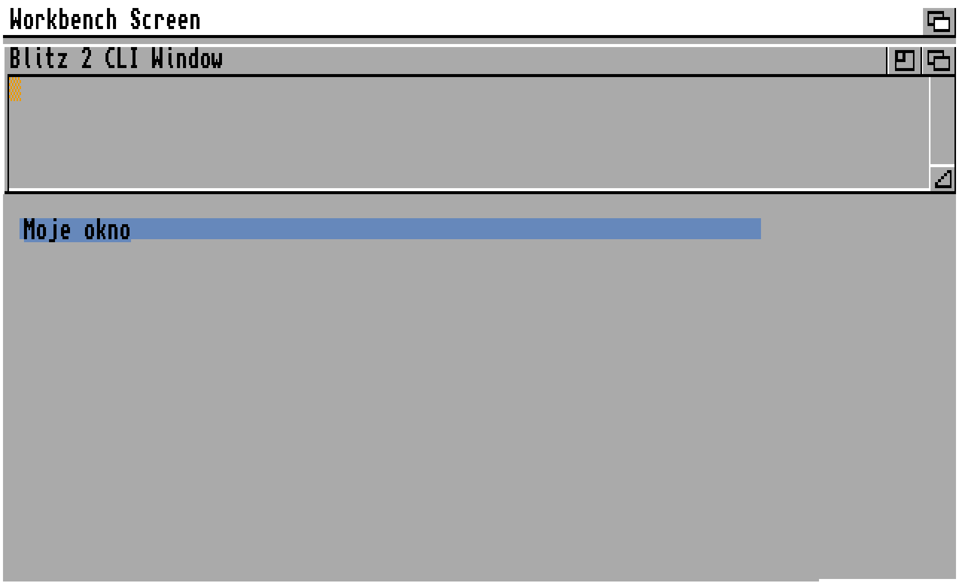


The same rule applies to other values from the list. Please note that the list also has options to specify how to treat the window frame and display the window on the active screen.

At the end of the line containing the WINDOW command, we add two additional arguments that set the window colors. The first argument determines the color which will be used to display the window title.

The second number is the color number used to draw the outer window frame. Thanks to this, we can adjust the window appearance to the color palette on the screen we use, as well as the specific functions of our program.

See how unusual it can look like the same window when we turn on the "flag" called BORDERLESS:



Now it is even difficult to know what is the size and exact position of the window. This gives you additional possibilities to design the interface of your program.

If you open several windows, you should specify which one you want to use before performing the operation. Otherwise, you can cause unpredictable results or the program will be stopped due to an error. Remember that the program can be handled by the user in a variety of ways and you have to foresee it when writing the program.

To set a window with a given number as current, just use the USE WINDOW command, giving the number next to it, for example, like this:

```
Use Window 1
```

Please note that some windows should have their own minimum and maximum sizes declared. This is mainly related to the content or arrangement when you use your own screen with strictly defined parameters. To limit the possibility of resizing, use the next command named SIZELIMITS. Here's how to use it:

```
SizeLimits 50,30,200,50
```

First, we give the minimum size in the horizontal and vertical, and then the maximum - in the same order. This line should be placed before the command causing the opening of a new window. Until you enter the next line modifying the limits, they will apply.

You can also put the word SIZELIMITS after opening the window, but then each new window will keep the same parameters. Of course, you can change it at any time and it all depends on the way your program works, the loops contained therein, conditional statements and other elements.

To perform an operation on a specific window, you should first activate it. Otherwise, the current object may be another window and your program will not work correctly. Fortunately, this is not a complicated operation, because just enter a line similar to the following:

```
Activate 1
```

which means activating the window with the given number. This command has no additional arguments.

Remember that if you share these options to the user, windows should not change their position for no apparent reason. On the other hand, when objects have permanently assigned sizes, you will surely come have to the need to modify parameters. Now you know how to do it.

Of course, you can move the windows on the screen and change their size. The following commands are used for this:

- WMOVE (like Window Move)
- WSIZE (like Window Size).

We discuss them together, because both require only two arguments. In the first case, they will mean the position on the screen where the window is to be located, and in the second - the desired width and height of the window. In both cases, the numbers indicate the pixels on the screen.

Therefore, the appropriate lines with the commands will look like this:

```
WMove 150,50
```

or

```
WSize 200,100
```

Of course, each window must be finally closed. The FREE WINDOW command is used for this purpose. You should type it along with the window number, for example:

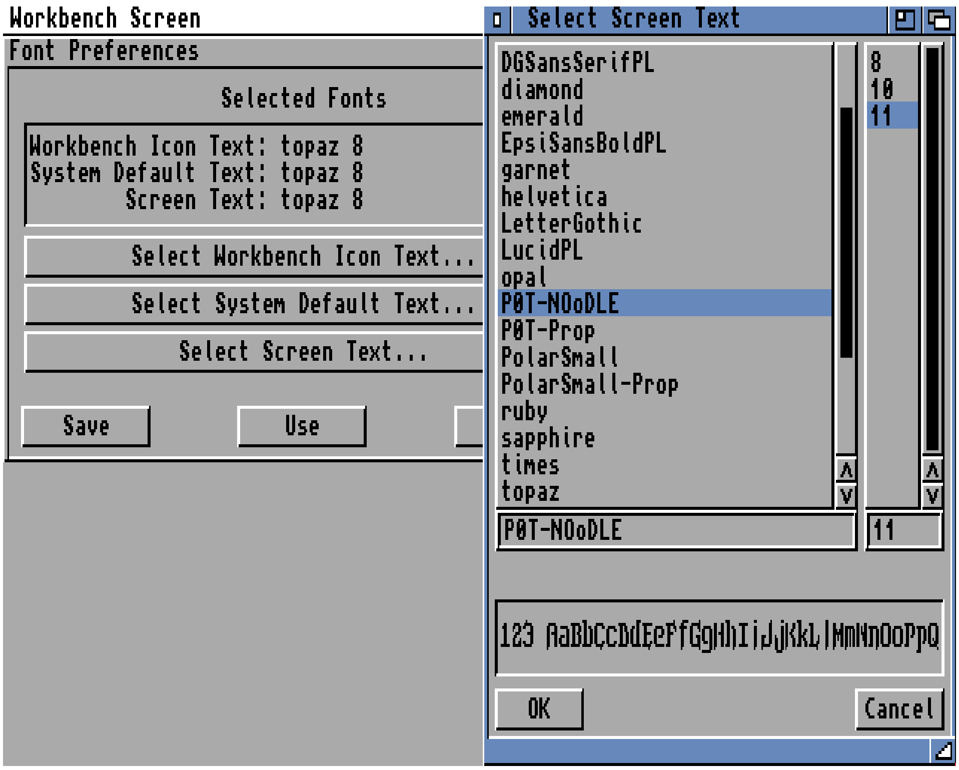
```
Free Window 1
```

Please note that the user's ability to close the window should be foreseen in the program in a special way, because it changes the way that the activities are performed. I will write about it later.

TEXT AND FONTS

In the window, you can display text to give various messages to the user. It is possible to specify the font to be used. When you use ready-made programs, you are certainly used to sharing fonts in the "Fonts" system directory.

If you open the font selection window, you can see all the names in alphabetical order, as in the "Font" preference program located in the "Prefs" directory on the system disk:


Workbench Screen
Font Preferences
Selected Fonts
Workbench Icon Text: topaz 8
System Default Text: topaz 8
Screen Text: topaz 8

Select Screen Text

DGSansSerifPL
diamond
emerald
EpsiSansBoldPL
garnet
helvetica
LetterGothic
LucidPL
opal
P0T-NOoDLE
P0T-Prop
PolarSmall
PolarSmall-Prop
ruby
sapphire
times
topaz

8
10
11

P0T-NOoDLE 11

123 AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQ

When writing your own program, you must first "read" the font. Of course, it means that the corresponding file must be located in the "Fonts" directory.

In other words, the user must know that he has specific font installed in the system before running your program, or you must write an installation script (preferably using the "Installer" program), which will do it automatically.

You can load fonts in Blitz Basic using the LOADFONT command, where you have to provide:

- font number,
- the name of the typeface you want to use,
- vertical size.

The entire line may look like this:

```
LoadFont 1,”opal.font”,12
```

Additionally, it is possible to automatically set the font style using the parameters at the end of the line. To do this, just enter a number denoting:

- **2**	- bold font,
- **4**	- italic font,
- **1**	- underline font.

In our example, it loads a 12-pixel typeface called "Opal". This font belongs to the standard Workbench files, so you can use it without the need to install additional software.

After loading the font, nothing prevents you from using it in the window. For this purpose the WINDOWFONT command is used, where this time you must enter the font number assigned to the typeface, for example:

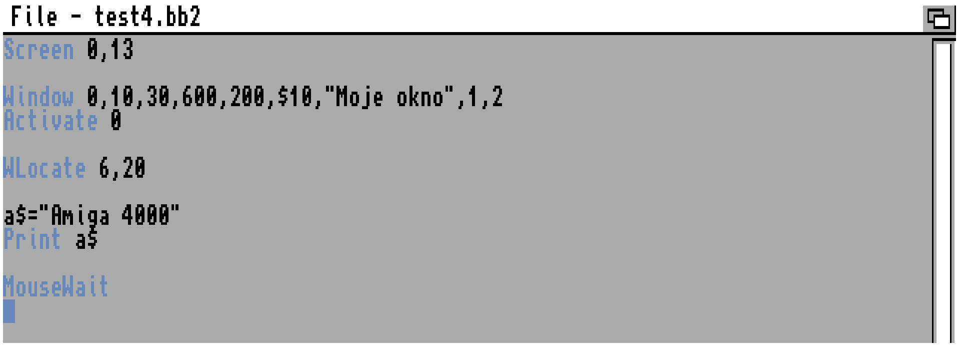
```
WindowFont 1
```

The number is obviously the same as the number entered in the line with the previous word LOADFONT, so you can precisely specify the character set used by the program interface.

Now you can print the text in the window confident that you will not use too small or too large a font. At the beginning, place the cursor in a specific position - usually it is the upper left corner of the window, although everything depends on your needs. To do this you need to enter a line containing the WLOCATE command, and give coordinates - horizontal and vertical.

Remember that the position is always determined using the points on the screen (in pixels), not the text line. In addition, each window has its own position of the text cursor, so the change in one window will not affect the position of the cursor in another window.

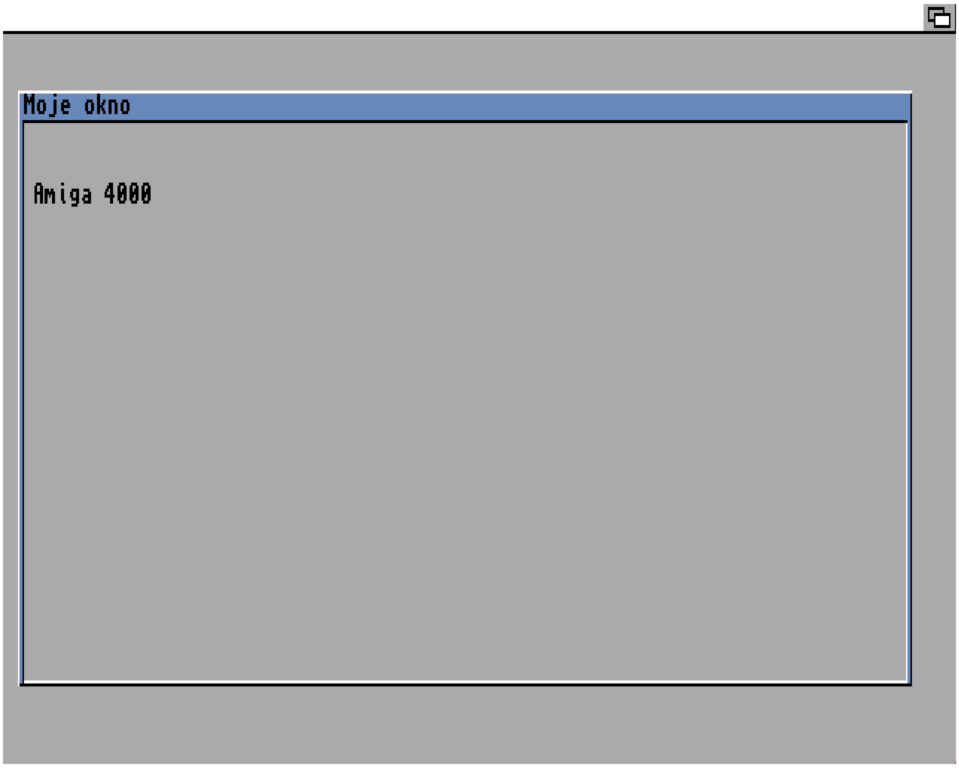
Here is a short program presenting these possibilities:



File - test4.bb2

```
Screen 0,13

Window 0,10,30,600,200,$10,"Moje okno",1,2
Activate 0

WLocate 6,20

a$="Amiga 4000"
Print a$

MouseWait
```

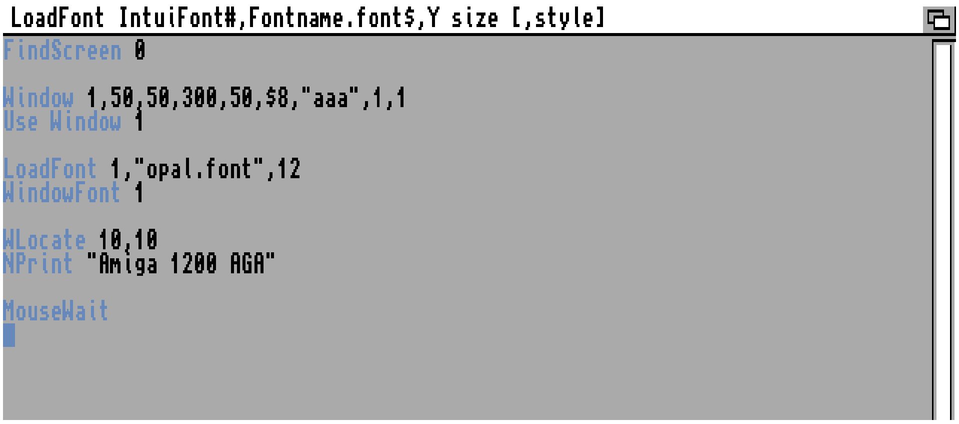
and the result in window:



Moje okno

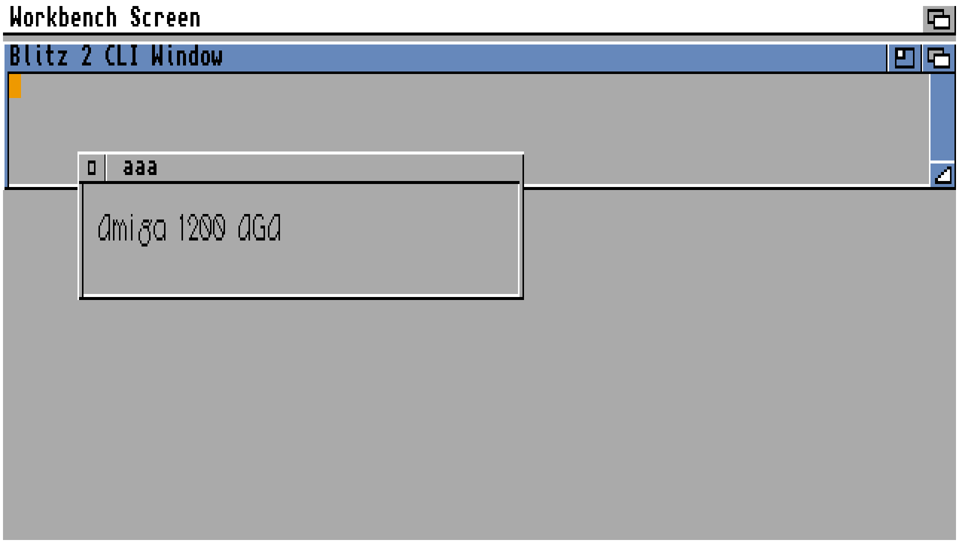
```
Amiga 4000
```

Another example changes the font and opens a smaller window with a close button. Look at the listing:



```
LoadFont IntuiFont#,Fontname.font$,Y size [,style]
FindScreen 0

Window 1,50,50,300,50,$8,"aaa",1,1
Use Window 1

LoadFont 1,"opal.font",12
WindowFont 1

WLocate 10,10
NPrint "Amiga 1200 AGA"

MouseWait
```

and the window, this time - inactive, but on the Workbench screen:



Note how many possibilities give only a few lines containing very simple commands. It turns out that for just a few minutes you can write a program that displays text in window open on the Workbench screen. This is, of course, only the basic way to use the system graphical interface, but it is worth noting that we do not use any non-standard functions, so our program is fully compatible with the Amiga operating system.

SPRITES

The next important element, especially when creating games, are so-called sprites. These are graphic objects that can be used, for example, to animate the hero in the game. They are created, displayed and moved on the screen separately from the other parts and independently of each other. This is possible because these objects are supported directly by the Amiga chipset. This means that they do not have to be removed manually when we want to make a hero's move, in addition, sprites do not damage the bitmap graphics. Using them, we can create complex effects on the screen.

Blitz Basic allows you to create sprites in 2 or 4 bitplanes, i.e. they can contain 3 or 15 colors. The basic width of the object is 16 pixels and any number of lines vertically. Sprites also require the setting of a separate color palette. Remember that Amiga has hardware support for 8 sprites.

- Creating sprites

To display the sprite, first load the graphic using the LOADSHAPE command. The given file will be treated as a "shape" with a specific number. For example, the following line:

```
LoadShape 1,″Work:data/graphic2.iff″
```

will load the "graphic2.iff" file and create the "shape" with number 1 from it. The file must be saved in the IFF ILBM format.

The next step is to assign the loaded file to the sprite. The GETASPRITE command is used for this, which requires two numbers, in order:

- target sprite,
- source "shape".

I emphasize this information, because it can be easily confused. So if you want to create a sprite with the number "zero" using our "shape" with the number 1, enter the following line:

```
GetaSprite 0,1
```

From this moment, the sprite is created in memory. It is independent of the "shape" that can be removed without losing information about the sprite. Thanks to that you will save memory. To do this, just use the FREE SHAPE command by entering the "shape" number, for example:

```
Free Shape 1
```

- Displaying sprites

When we have a sprite we can display it on the screen. We will get this using the next word SHOWSPRITE, which has a more complex syntax.

Take a look at an example:

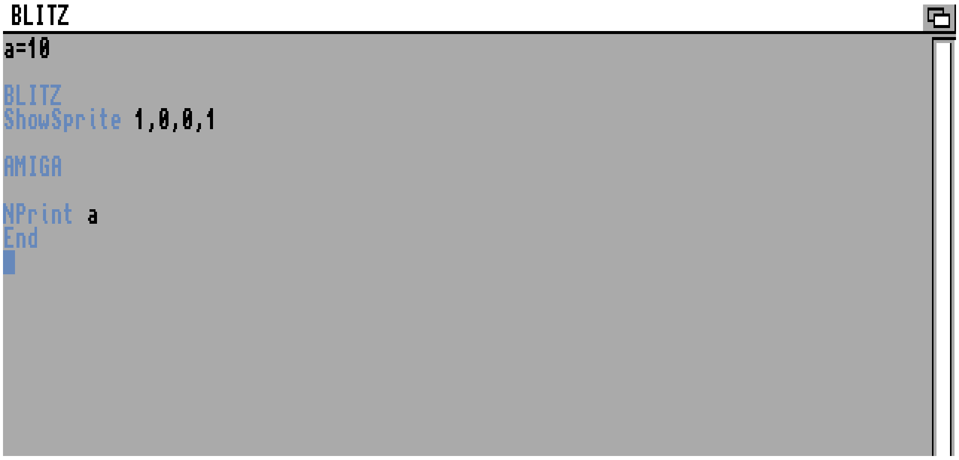
```
ShowSprite 0,50,30,1
```

The first number is the number of the sprite that was created. The next two are coordinates - horizontal and vertical, where you want to display the object. The last value is the so-called "channel" to which the sprite will be assigned. We have 8 channels, because - as I have already mentioned - this is how Amiga handles hardware sprites. The coordinates will refer to the Lowres display mode.

Please note that these functions are only enabled in the so-called "Blitz" mode when we turn off the Amiga system and have direct access to chipset. To start it, just enter the name in a separate line:

```
BLITZ
```

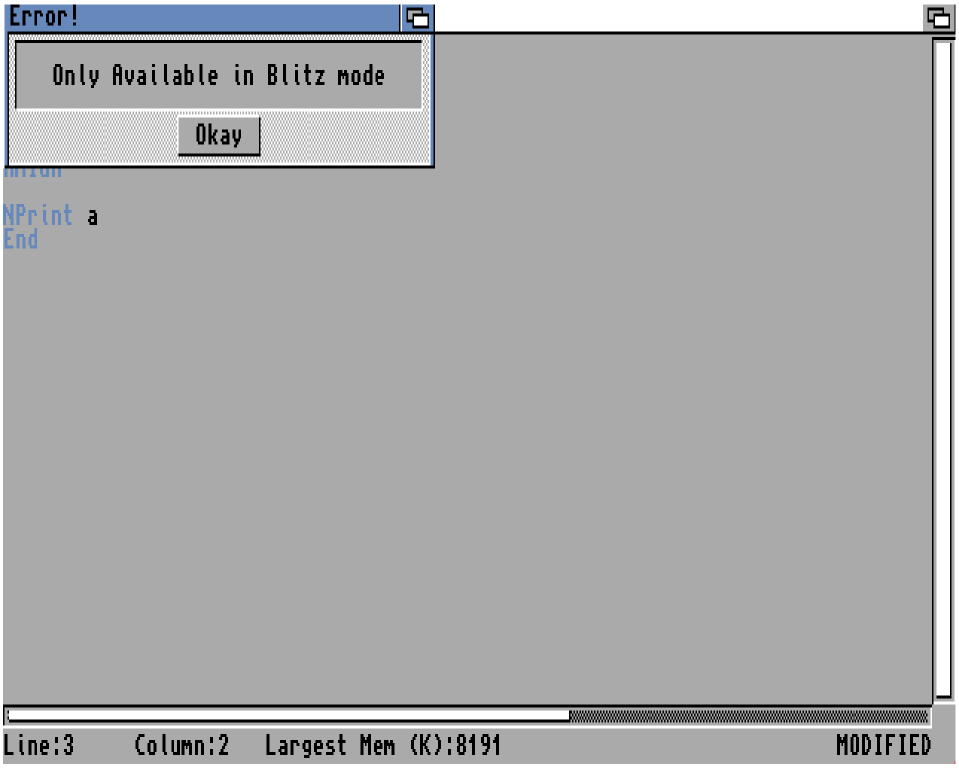
To return to the "normal" state, use the "AMIGA" command. For simplicity look at the illustration:



```
BLITZ
a=10

BLITZ
ShowSprite 1,0,0,1

AMIGA

NPrint a
End
```

About the "direct" operating mode we will write more later. At the moment you should to learn basic commands and their functions.

However, if a window like the following appears on the screen:



```
Only Available in Blitz mode
```

```
NPrint a
End
```

```
Line:3      Column:2    Largest Mem (K):8191                    MODIFIED
```

this means that you must enable the "Blitz" mode.

- Size of the sprites

I wrote earlier that the standard width of sprites is 16 pixels. If you want to use larger objects in the program, you must change the display mode using the SPRITEMODE command along with the following numbers:

- width: 16 pixels	0
- width: 32 pixels	1
- width: 64 pixels	2

You can save the finished object to a disk in a separate file. For this purpose, use the word SAVESPRITES by entering the range of sprite numbers and the file name. For ease of use, see how it can look like:

```
SaveSprites 0,2,"Work:data/my_sprites"
```

The two values at the beginning define the start and end numbers of the sprites to be saved in the file:

```
my_sprites
```

Of course, you can also save one object, but storing information about multiple sprites in one file saves the space on the disk. This is especially important when you want to run your program from a floppy disk on unexpanded Amiga. The file is read into memory using the analogous LOADSPRITES command. Its use is the same, also the range of numbers and the file name should be specified. For our file it will take the form of the following line:

```
LoadSprites 0,2,"Work:data/my_sprites"
```

Please note that you do not have to read all saved items right away, which allows you to save space in your computer memory. You can also remove any object from memory using the word FREESPRITE. Just enter only the number of the sprite.

- Changing the display order

An important feature of the sprites is the ability to display "at front" or "behind" bitmap graphics visible on the screen. This is enabled by the INFRONT command, where must be specified one of the even channel numbers to which the object is assigned, for example:

```
InFront 2
```

After executing the above line, sprites placed on channels equal or higher than 2 will be displayed "behind" graphics. Other objects - on the channels below the number 2 - will be shown "at front" the graphics. In our case it will look like this:

- channels: 0, 1, 2	- sprites displayed "at front",
- channels: 3, 4, 5, 6, 7	- sprites displayed "in the back".

The display ordering of the objects looks a bit different in the mode called Dual Playfield. This is another broad topic, which is why we will write about it later.

ANIMATION

Blitz Basic allows you to handle not only IFF graphic files, but also animation, which is popularly called the "ANIM" format. It must be added that this format has many variations, while in your program you can use the so-called ANIM-2, or "Long Delta Mode". This is the format used by popular graphics editors, for example the famous "Deluxe Paint".

If you want to use the animation in Blitz Basic, you need to check what is format of the file and, if necessary, perform the conversion or save the data again using other parameters.

- Loading files

To load an animation, you should use the LOADANIM command, which requires a number and file name. So it can look like this:

```
LoadAnim 1,"Work:files/data5.anim"
```

Playback is just as simple, because it will start after execution the line similar to the following:

```
InitAnim 1
```

You have to enter the same number. Previously we must also open the screen in the appropriate resolution. How to do it we write in the section entitled "Screens".

However, the above command will only show the first two frames of the file. If you want to continue playing, you must use the command named NEXTFRAME. In this case you also need to enter only the number assigned to the animation.

Please note that thanks to this you will see one next frame, so for a smooth playback you need to make a longer loop. It does not matter in what form it will be saved, it all depends on the other parts of your listing.

- Reading file parameters

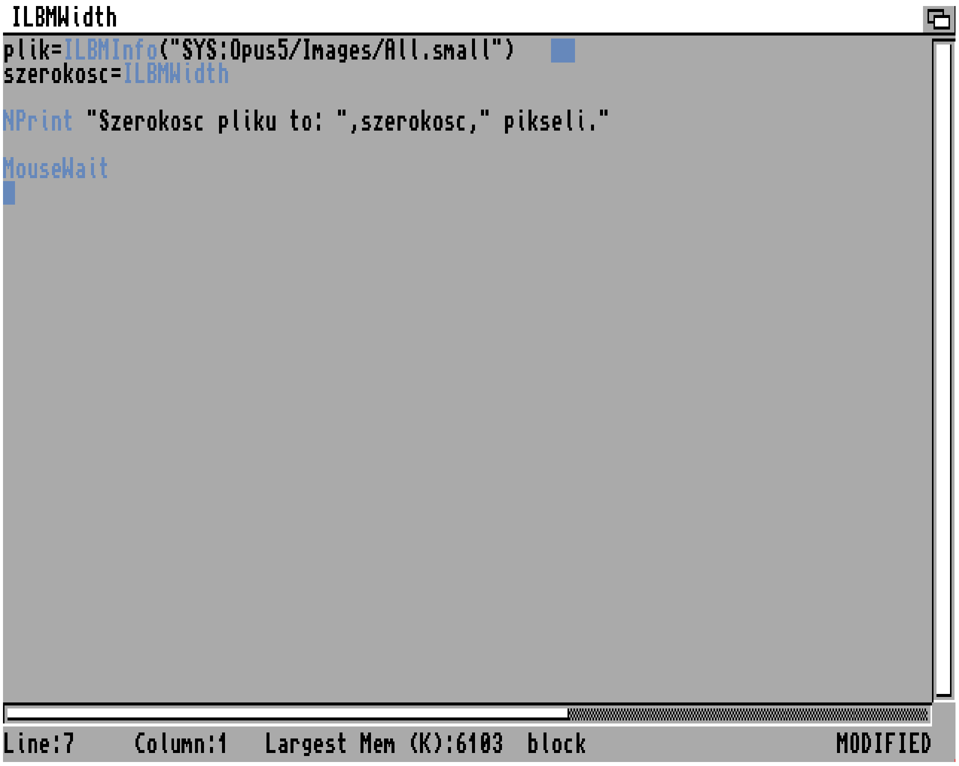
When the program reaches the last frame of the file, it will automatically "jump" to the beginning, that is again to the first two frames. You will not always want to get this effect, but it does not necessarily have to involve recognition the specific length of the animation.

You can read information about the contents of an ANIM file in several different ways. The first is to use the FRAMES() function, so you can check the amount of all frames in the file. To do this, create a new function, for example:

```
a=Frames(1)
NPrint a
```

On this base, you can easily write a program that checks the number of frames saved in a file and plays a specific range, entered by the user.

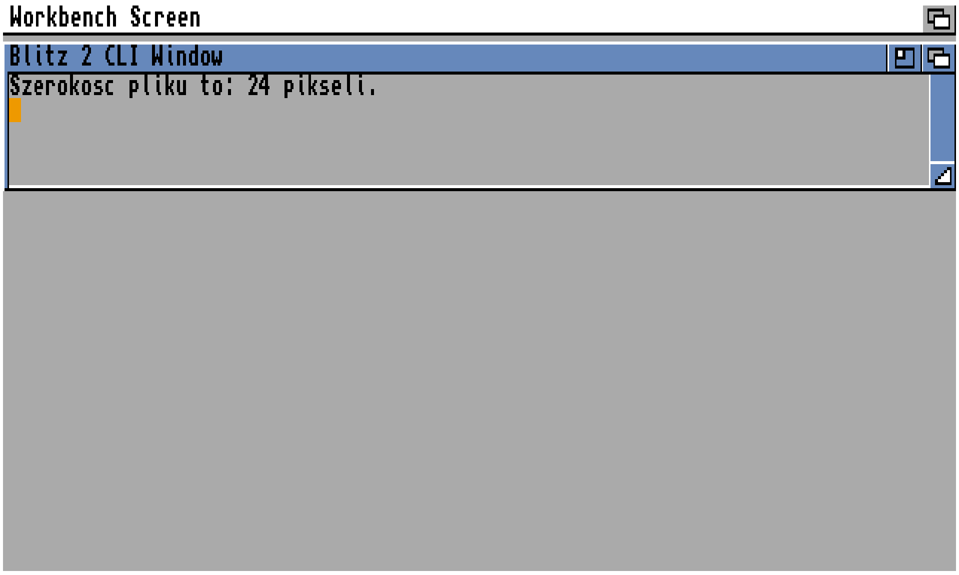
Another option is to use a group of functions with names beginning with the "ILBM" symbol. Thanks to the ILBMINFO() function, you can get detailed information about the file. Here's an example:



```
plik=ILBMInfo("SYS:Opus5/Images/All.small")
szerokosc=ILBMWidth

NPrint "Szerokosc pliku to: ",szerokosc," pikseli."

MouseWait
```

```
Line:7     Column:1    Largest Mem (K):6103  block                 MODIFIED
```

and the result of action:



```
Szerokosc pliku to: 24 pikseli.
```

As you can see, the first function reads the information about the file and the second one gives specific parameters without the need to enter any parameter. Other features of the file can be read separately, using one of the following functions:

- ILBMWIDTH(),
- ILBMHEIGHT(),
- ILBMDEPTH().

The method of use is analogous. Thanks to them, you can check the width and height of the graphics that make up the animation and the "depth", which is the number of bitplanes.

- Display mode

The last option is to read the graphical mode of the animation, which can be especially important, if you want to play the file on your own screen, instead on window on the Workbench. You will have to use it in a specific display mode and different amounts of colors.

To make this possible, automatically use the ILBMVIEWMODE() function before starting playback. Insert the conditional statements that open the appropriate screen, depending on the data obtained. The function can pass the following values:

- Lores mode	0
- Hires mode	32768
- Interlace mode	4
- Extra Half-Brite (EHB) mode	128
- Hold-And-Modify (HAM)	2048

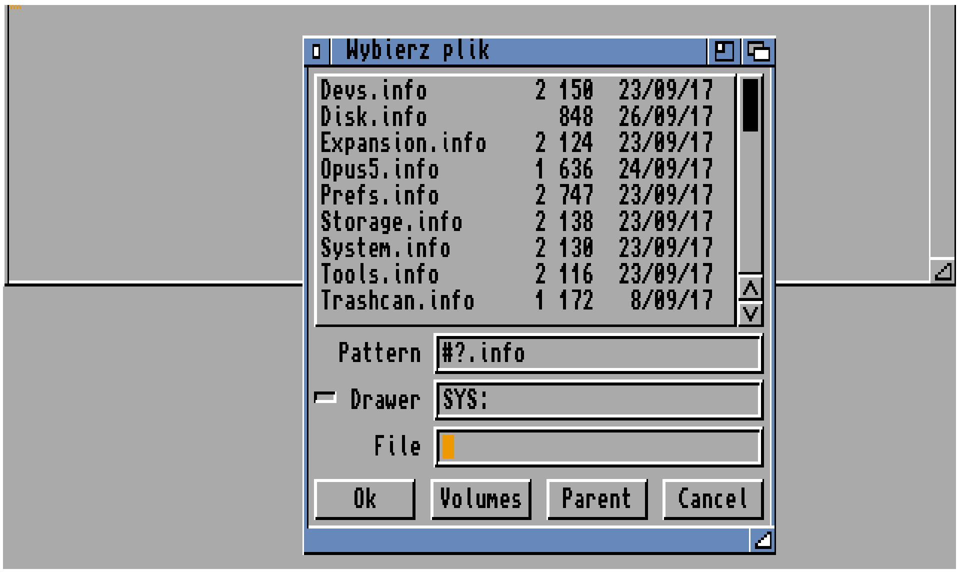
According to these numbers - or a combination of them, for example Hires and Interlace, you should open new screen to make the animation play correctly.

SELECTION WINDOWS

The file selection window is related to the pull-down menu discussed in the previous sections. In almost every program you can load and save files, so your program can not be deprived of this features. The easiest way to create such a window is to use the AmigaDOS function, specifically the REQUESTFILE command. To display the file selection window you need to set the basic options. Take a look at an example:

```
requestfile DRAWER="SYS:" TITLE=„Select file"
PATTERN="#?.info"
```

As a result of this line, you will see the following selection window:



```
Wybierz plik
Devs.info          2 150  23/09/17
Disk.info            848  26/09/17
Expansion.info     2 124  23/09/17
Opus5.info         1 636  24/09/17
Prefs.info         2 747  23/09/17
Storage.info       2 138  23/09/17
System.info        2 130  23/09/17
Tools.info         2 116  23/09/17
Trashcan.info      1 172   8/09/17
```

Pattern #?.info
Drawer SYS:
File

The title has been given in quotation marks immediately after the name of the argument named TITLE. The PATTERN argument allows you to determine which items will be displayed in the window. In our case, we want files with the extension "..iff" to be visible on the system disk, so the symbol ("SYS:") we typed as the argument DRAWER.

In this way, we can cause that the user will be able to change the work of our program. Many useful elements can be created on the basis of the mentioned elements.

The second way to create a selection window is to write a short program that will be independent of other elements of the operating system. It does not require a lot of effort, because just open the window, create a list of files in a specific directory, and then read the information on the choice made by the user.

Such a method is not recommended, because your program will at this time create an unusual element, instead of displaying the standard system window. On the other hand, you can create your own module for handling disk operations without restrictions imposed from the outside.

In my opinion, you should stick to the guidelines for writing programs, so use usual windows that can be found in other programs. So we will use system libraries.

Therefore, you need to know the FILEREQUEST$() function and the associated group with names beginning with the "ASL" symbol. Of course, this name comes from "asl.library" file, which is part of the Workbench.

The FILEREQUEST$() function allows you to open a standard window for selecting files on the currently used screen. The program will be paused until the user selects the file or the "Cancel" button. If the file is selected, the function will result with full name of the file, in the form of a string. When the operation is canceled, you will get an empty string. A line that uses this function may look like this:

```
a$=FileRequest$("Select file",b$,c$)
```

As you can see, it should be assigned to a text variable. I discussed it in the chapter "The first program" and this is not unexpected element. In brackets you have to enter three parameters in the following order:

- window title (here - "Choose file"),
- file path (here - variable "b$"),
- file name (here - variable "c$").

The title may be any string of characters, but it should indicate to the user what option has been called. It may happen that the file saving function will be selected instead of loading. Therefore, the window should have a precisely defined title. You can also use the additional variable instead of typing specific content.

We enter the name of the path so that it can be automatically read after the window is displayed. This allows you to see the list of files from a specific directory at once. The user can change it later. You do not have to give the path, but it is often convenient, because it is also possible to lead on the user to directory where your program has been saved while installation.

Please note that it must be a text variable and its maximum length is 160 characters. This should be sufficient for most applications, although the maximum file length in the Amiga system is only 107 characters. For this reason, your program should not create long prefixes or suffixes to file names, and file extensions.

The file name - as in the case of the path - is also a text string, but the maximum length in this case is 64 characters. Blitz Basic has a limitation, but it still exceeds twice the capabilities of the standard file system called "Fast File System" (31 characters). Of course, you can also use other file systems on the disk, but it is not possible on every Amiga model. In addition, this is not the topic of this book.

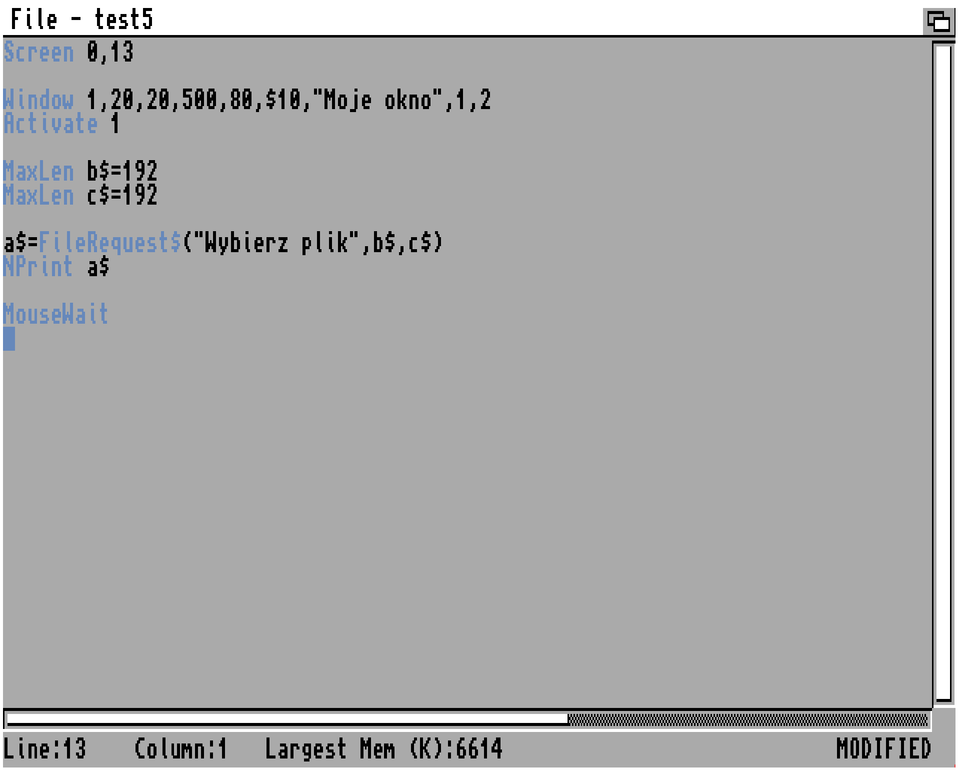
Before you call the FILEREQUEST$() function, you must also specify the maximum size of the variable denoting the path and the file name. In our example, these will be "b$" and "c$" variables. To set the size, use the MAXLEN command in the following way:

```
MaxLen b$=192
```

and

```
MaxLen c$=192
```

The above values may be different, but they should not be too small, otherwise the program will not be able to launch. The MAXLEN command is used to allocate a certain amount of memory for variables, so remember to use it before declaring them. In a larger program, this may look like this:



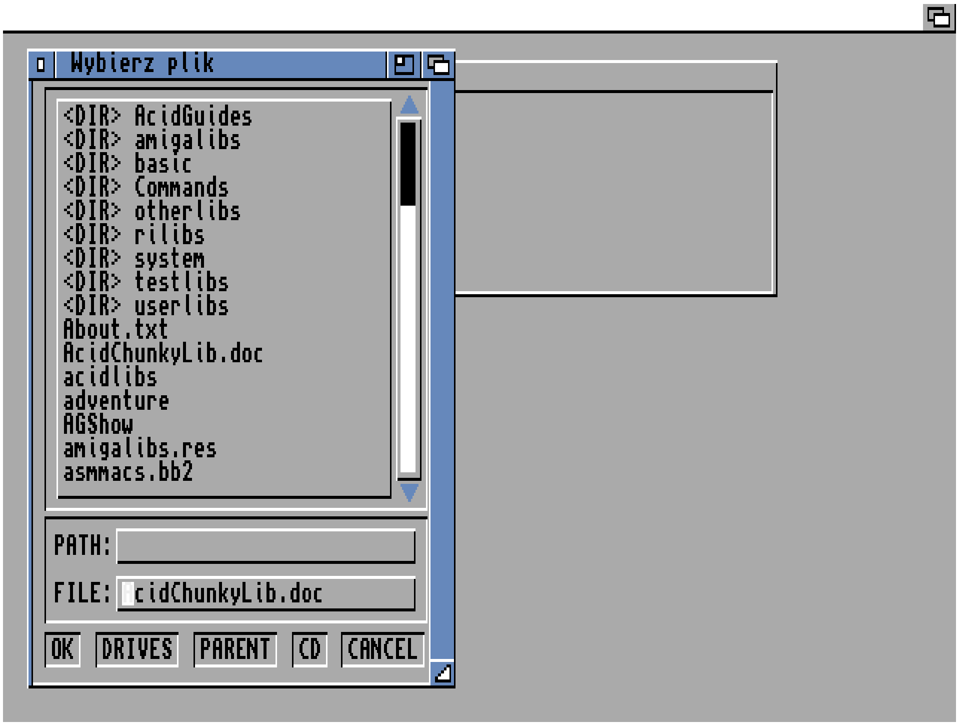
```
Screen 0,13

Window 1,20,20,500,80,$10,"Moje okno",1,2
Activate 1

MaxLen b$=192
MaxLen c$=192

a$=FileRequest$("Wybierz plik",b$,c$)
NPrint a$

MouseWait
```

```
Line:13    Column:1    Largest Mem (K):6614                    MODIFIED
```

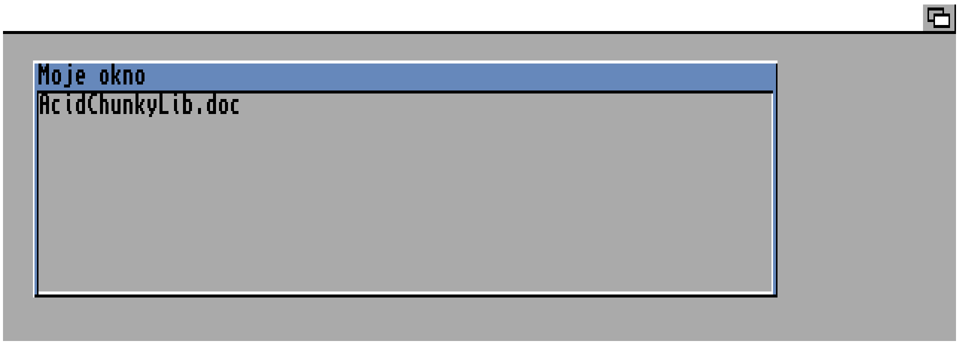
This is the first part of the program. The creation of the function itself will not call the selection window, you have to enter the line with the NPRINT command by providing the variable assigned to the function. For example, as in our picture, that is:

```
NPrint a$
```

You can enter the above line directly after the declaration of the variable "a$". See how the result of such a program looks like:



In the illustration we have already indicated one of the files visible in the selection window. When you select the "OK" button, the name will be printed in the window placed in the background:



JOYSTICK

When writing a game, you can not skip the joystick. In the utility programs using individual user interface solutions or creating different working areas, the ability to control the mouse pointer will be useful.

By using the JOYX() and JOYY() functions, we can read the joystick state. Both should be used in the same way, that is - as usual with the function - assign them to a numeric variable, for example:

```
a=JoyX(1)
```

As a parameter, we give the port (0 or 1) where the joystick is connected. In Amiga, the joystick is usually used in port with number 1, but there is no problem to use "zero" or both at the same time - for two controllers.

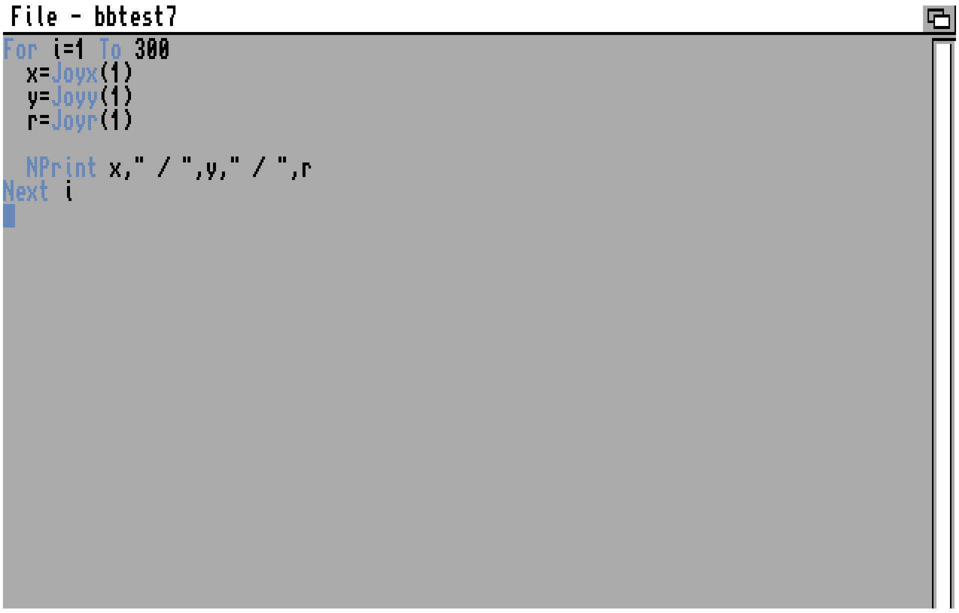
As a result of this function we get the following values:

- no direction	**0**	(JOYX, JOYY),
- left	**-1**	(JOYX),
- right	**1**	(JOYX),
- up	**-1**	(JOYY),
- down	**1**	(JOYY).

The joystick status can also be read within one function called JOYR(). The usage is the same, but this time we get values from 0 to 8, which means:

- up	0
- up and right	1
- to the right	2
- down and right	3
- down	4
- down and left	5
- left	6
- up and left	7
- no direction	8

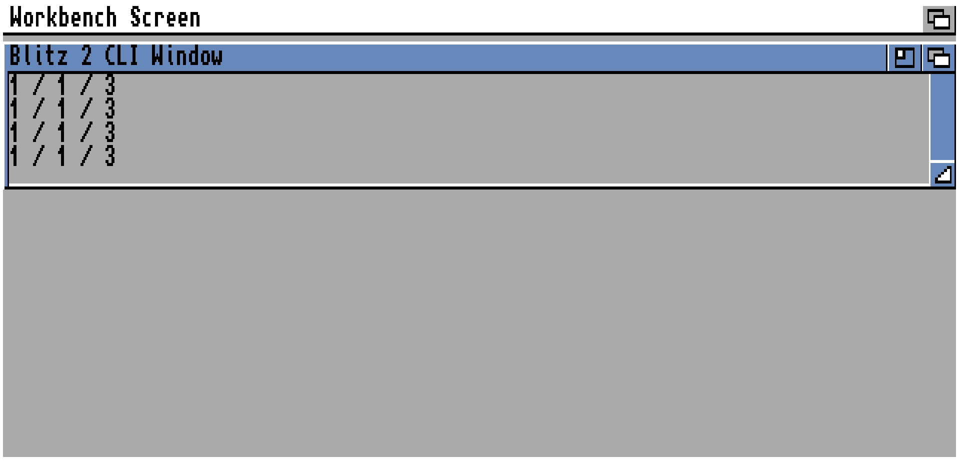
In this way, you can check whether the user of your program has used the joystick. Remember that testing the state must be placed in the loop, as well as other important elements. Look at another simple example in the editor:



File - bbtest7

```
For i=1 To 300
  x=Joyx(1)
  y=Joyy(1)
  r=Joyr(1)

  NPrint x," / ",y," / ",r
Next i
```

and the result of its operation:



```
1 / 1 / 3
1 / 1 / 3
1 / 1 / 3
1 / 1 / 3
```

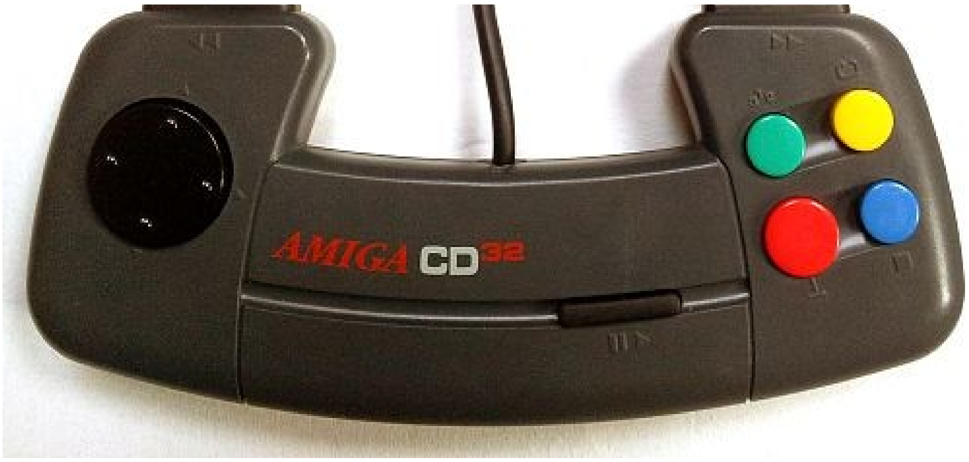
In addition to directions, it is also important to use the Fire button. The function JOYB() is responsible for this, the name is an abbreviation of Joystick Button. As an argument, we give the port number again, that is:

```
b=JoyB(1)
```

but this time you should expect other values:

- no Fire button is pressed	**0**
- pressing the Fire button	**1**

In Blitz Basic you can also program the joypad control known from the CD32 console. Keep in mind that it has additional buttons, as in the next picture:



The operation of additional buttons requires the use of another function called GAMEB (that is Game Button). Its usage is still the same, but you will get the values as below:

- Play / Pause	1
- Reverse	2
- Forward	4
- Green button	8
- Yellow button	16
- Red button	32
- Blue button	64

Remember that the directions are marked with the previous functions, and GAMEB is only used to check the buttons not available on the regular joystick.

Various controllers can be connected to the Amiga CD32, so if your program uses all possible functions, it can decrease the group of your recipients. The player will have to use the original joypad or replacement with additional buttons. Therefore, I suggest that you add the same options supported with the keyboard or mouse - then the program will work universally on every Amiga.

MOUSE

The mouse handling is more extensive, because concerns the position of the indicator visible on the screen. You have certainly seen programs where the appearance of pointer is changed. It is also used in games, although less frequently, because they are usually based on joystick support.

- Movement tracking

The first thing is determine whether the program is to track the mouse movement. The MOUSE command is used to enable this service:

```
Mouse On
```

or disable it:

```
Mouse Off
```

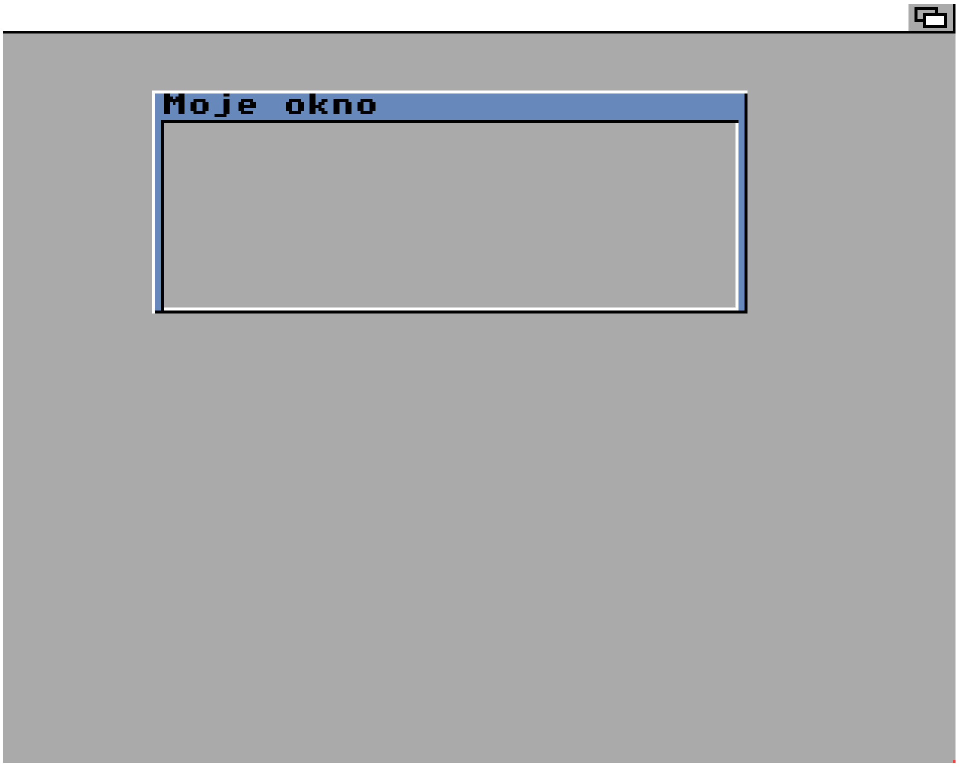
By default, the pointer can be moved around the entire screen, which for Lowres mode gives the area with starting coordinates of 0,0 and final 320,200 pixels. If necessary, you can limit it, just use the MOUSEAREA command. It accepts four arguments:

- initial horizontal coordinate,
- initial vertical coordinate,
- final horizontal coordinate,
- final vertical coordinate.

The example line may look like this:

```
MouseArea 50,30,200,75
```

Now the user will be able to move the pointer to a smaller area, i.e. in the box with the coordinates presented in the following window:

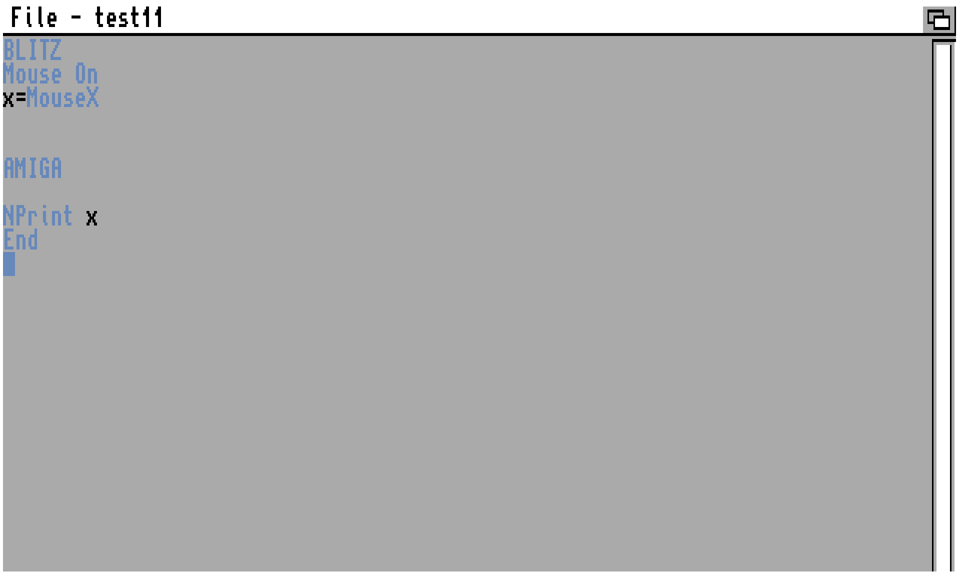


- Pointer location

The operation of the program can depend on the position of the mouse pointer. Such a possibility is provided by two functions - MOUSEX() and MOUSEY(). They read the current coordinates and, as before, they should be placed in a loop so that the program can react appropriately to the change of position.

The above functions are possible to run only in the so-called "Blitz" mode, which is characterized by disabling the operating system. You have direct access to the Amiga chipset, so we can say in short that it is a "non-DOS" way of writing programs.

To start the "Blitz" mode, just enter this name on a separate line. For example, as in this simple listing:



```
BLITZ
Mouse On
x=MouseX


AMIGA

NPrint x
End
```

The screen will be blanked, so you have to write programs that use this function very carefully. Back to the "normal" mode of operation is carried out by entering the word "Amiga", because this is the name of second mode. And it can be seen in our illustration.

I will write about operate your computer in "Blitz" mode later, at the moment it is important that you are aware of the existence of two mentioned modes.

Please note that the pointer status is updated 50 times per second, so the results will be very precise and will often change. This is also to be foreseed when writing a program.

If you want changes to be made when moving certain zones on the screen, you should use several conditional statements.

Blitz Basic also gives you the opportunity to get the mouse position in relation to various elements on the screen. The first option is to read the location considering the upper left corner of the current screen as the starting point. This can be useful if you change positions of several open screens. For this purpose, there are two functions with names derived from English words Screen Mouse:

- SMOUSEX(),
- SMOUSEY().

They should be used in the same way as MOUSEX() and MOUSEY(), but remember that depending on the position of the screen, you can get different values.

The next functions concern the position of the pointer in the currently active window. Therefore, they have the names WMOUSEX () and WMOUSEY (). These values determine the position in relation to the upper left corner of the window, not the screen.

In addition, they may include the information about window frame size or only the content itself. It depends on the parameters of the window, specifically the GIMMEZEROZERO flag.

When this flag is not active, the given value will point to the upper left corner of the window frame. Otherwise, it will mean the upper left corner of the window content.

- „Events” testing

The position of the pointer can also be read when a so-called "event" occurs in the window. Again, you should use the specific functions - in the same way - but this time they have different names:

- EMOUSEX(),
- EMOUSEY().

Please note that events do not happen all the time, like 50 times a second earlier, so the loop containing the above words will work faster and have to check the actions in a different way. I will discuss these possibilities later.

- Pointer speed

In addition to the changing of mouse pointer location, it can be moved at different speeds. You can also handle this feature, because you have two more functions:

- MOUSEXSPEED(),
- MOUSEYSPEED().

The first one is responsible for checking the horizontal motion, the second one - vertically. In this case, you can get positive or negative numbers, depending on the direction of the pointer movement. The functions will indicate the following changes:

- move left	negative value of MOUSEXSPEED(),
- move right	positive value of MOUSEXSPEED(),
- move up	negative value of MOUSEYSPEED(),
- move down	positive value of MOUSEYSPEED().

The higher the value, the higher the movement speed. Remember that the pointer may have a various appearance, it may also not be visible on the screen. How to change it we write in the section titled "Changing the appearance of mouse pointer".

The above functions will read the status regardless of whether the movement leaves changes on the screen or whether the user can not follow the pointer.

It is also worth emphasizing that before using the functions you must use the another line:

```
Mouse On
```

Otherwise, they will not react to changes.

- Buttons handling

The mouse buttons state testing should be performed using the same function as for the joystick, i.e. JOYB(). Previously, I wrote that it takes the value 0 or 1. In the case of a mouse, it will be the following numbers:

- pressing the left mouse button	**1**
- pressing the right mouse button	**2**
- pressing both buttons simultaneously	**3**

As you can see, we can do it in various configurations, depending on your needs. Again, everything should be located in a loop, such as:

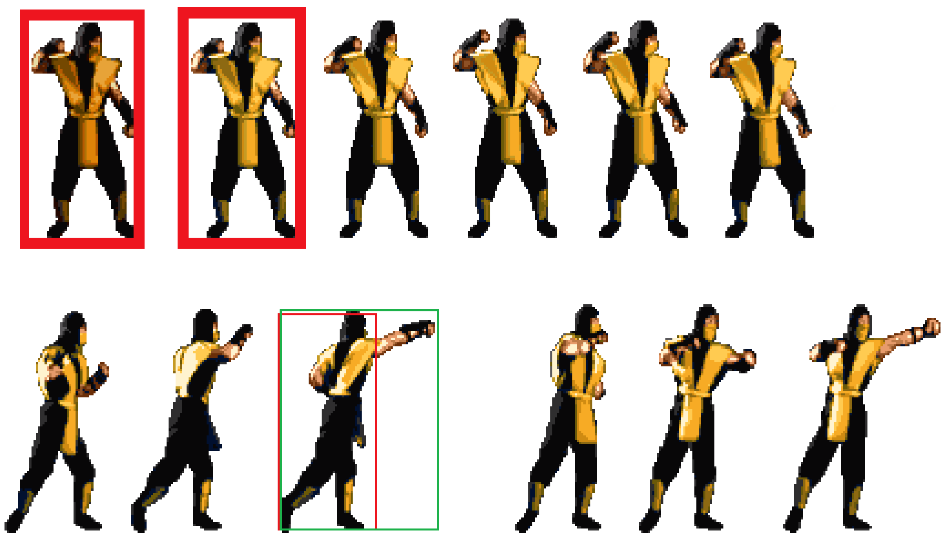
```
While Joyb(0)=0
    ...
Wend
```

These lines will make the program wait for pressing any mouse button to end the loop.

COLLISIONS

When you want to write your own game, you will have to handle the interactions between objects. Your hero may be faced with an enemy and lose some energy. The contact of two objects on the screen is called "collision". It is a situation when the shapes used to create them have common parts.

To detect collisions, we usually use simplified versions of objects depending on the needs. The more we apply simplification, the faster will be the detection of collisions, although it is a bit less precise action. To understand it better, look at how this may look like for the hero of a popular game:



Of course, the player does not see the technical aspect of collision detection, so you have to take care of the proper reaction of the program. This is especially important when we have a large number of animated objects on the screen.

- Sprites and graphics

In Blitz Basic you have various ways to detect collisions. The easiest way is to call the reaction when the sprite contacts the specific color of bitmap graphic on the screen. It's about finding a "dangerous" color that will trigger a reaction if the sprite reaches the same position as the color. Thanks to this, it is possible to design zones where the hero can move, just like in a typical arcade game.

To make this possible, you must use the SETCOLL command. In the line you should give the color number and the "depth" of the screen (that is the number of bitplanes) that will be tested when detecting the contact. E.g:

```
SetColl 2,3
```

This will make that the "hazard" color will be the second bitplane of four available, i.e. 8 colors of graphics. This method allows you to control collisions quite closely, regardless of the actual number of colors available on the screen.

Similarly, you can cause each odd color in the palette to be a "dangerous". In other words, they will be numbers 1, 3, 5, 7, etc. In this case, just use the word SETCOLLODD without any argument.

As part of the color manipulation operation, the SETCOLLHI command is still available. Thanks to this, dangerous colors will be the specified range of colors - the second half of the palette. This will happen after giving the "depth", i.e. for example:

```
SetCollHi 4
```

The above line means that we take into account 4 bitplanes, that is 16 colors. In this case, the second half of the palette are the colors from the numbers 8 to 15. Note that giving the number of colors is similar to previous feature, but now the function will assume a range of colors instead of specific items.

In order for the collision to be detected, it is not enough to determine how the program works. At some point, you must use a command so that the state of the objects can be checked. The word DOCOLL is used for this purpose. Remember that you must type it in the program only after using the previous commands, otherwise it will not work.

However, the DOCOLL command itself only causes the collision to be tested. After that you should specify that we want to check the contact between the sprite and the graphics. So you have to enter the next line containing the function PCOLL(), for example:

```
k=PColl(1)
```

The value in brackets is the number of the so-called "channel" where a specific object can be assigned. Thanks to that, you will indicate the sprite you want to call. More about the channels you can read in the section titled "Sprites".

If a collision is detected, the function will receive the boolean value TRUE (-1), otherwise FALSE will be passed (0 - zero). Remember that in addition to the unstandard using, the function behaves analogously to other ones, which we discussed in separate parts of the book.

- Collisions between sprites

In addition to testing "safe" and "dangerous" colors, you can check the contact between the different sprites. To do this, you must use the SCOLL() function, similar to PCOLL() from the previous section. In this case we provide two different channel numbers as parameters.

The goal is the same - we define the sprites to be taken into account during the collision. For example, it may look like this:

```
k=SColl(1,3)
```

As before, we get the logical TRUE or FALSE.

You can also check whether the two areas occupied by the sprites overlap each other. To do this, we use the SPRITESHIT command by typing:

- number of "first" sprite,
- horizontal coordinate,
- vertical coordinate,
- number of the "second" sprite,
- horizontal coordinate,
- vertical coordinate,

If the sprites overlap each other, the function will get the logical value TRUE, otherwise - FALSE.

SOUND

Audio support is an important issue in many programs. By default, you can use IFF files, or more accurately - a variation called 8SVX. It is a format for storing 8-bit sound data, both mono and stereo.

To play any sound, you must first load it into memory. We do this with the LOADSOUND command. Next you need to type the sound number and file name, preferably along with the full path. For example, like this:

```
LoadSound 1,"Work:barrel"
```

After this operation, you can enter an instruction to play the sound. Just enter the next SOUND command, but now you should give not only the number, but also the so-called "mask". It takes values from 1 to 15 and indicates which Amiga channels will be used for playback. Recall that we can use 4 sound channels, 2 for each stereo channel.

Therefore, the correct line may look like this:

```
Sound 1,15
```

This command will play the sound with number 1 on all Amiga channels.

It is possible to use all combinations of "switching on" and "switching off" the channels, just use appropriate number denoting the mask. Here is a list of possibilities:

Mask	Channel 0	Channel 1	Channel 2	Channel 3
1	enabled	disabled	disabled	disabled
2	disabled	enabled	disabled	disabled
3	enabled	enabled	disabled	disabled
4	disabled	disabled	enabled	disabled
5	enabled	disabled	enabled	disabled
6	disabled	enabled	enabled	disabled
7	enabled	enabled	enabled	disabled
8	disabled	disabled	disabled	enabled
9	enabled	disabled	disabled	enabled
10	disabled	enabled	disabled	enabled
11	enabled	enabled	disabled	enabled
12	disabled	disabled	enabled	enabled
13	enabled	disabled	enabled	enabled
14	disabled	enabled	enabled	enabled
15	enabled	enabled	enabled	enabled

By following the table, you will make the sound play on certain channels. In addition, you can set the volume of each one. To do this, add values from 0 to 64 at the end of the line. There must be the same amount as much the channels will be used for playback. For example, if you enter a line:

```
Sound 0,10,32,16
```

then two channels will be used - the second and fourth, so we added two arguments - numbers 32 and 16. This is not too intuitive to use, but in practice requires only checking the number of channels and entering a specific amount of additional arguments.

You can also change the volume during playback. Thanks to this, it is possible to create the effect of muting sound, amplifying or other, depending on the needs. To do this, we will use the next command named VOLUME. Like before, you must give the mask and the volume values.

So we can say that we write the same as for the SOUND command, but without the first argument. Instead of this:

```
Sound 0,10,32,16
```

use following line:

```
Volume 10,32,16
```

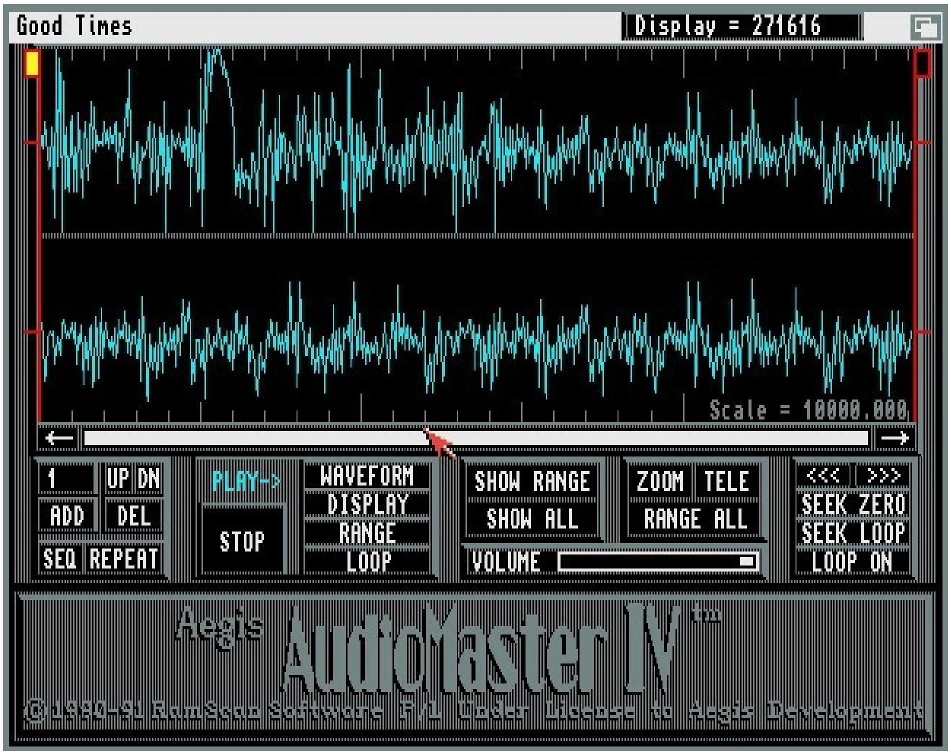
Please note that this time you do not change the sound file, but only modify the volume of the channels, so you can do it in different combinations.

Normally, the sound is played only once. You can use a loop to repeat it, but it's more convenient to enter the word LOOPSOUND. Its arguments are similar to SOUND, so you can change the volume at the same time when you start playing "in the loop". Here is an example:

```
LoopSound 0,10,25,42
```

The IFF 8SVX file can be used to store a starting point for audio loop. It can be created in sound samples editors, for example in the famous "AudioMaster". We especially recommend his fourth edition.

In the program, the loops can be created graphically using markers, which are normally placed at the beginning and end of the file. It looks like this:



You will not always have this option, moreover you may want to change the way you loop the sound. This is possible using the INITSOUND command, but the syntax is more complicated.

Here are its arguments in the correct order:

- sound number,
- sound length,
- pitch of sound,
- locations of loops.

Only the first two words are absolutely necessary for action, so you can simplify your work by entering a line similar to the following:

```
InitSound 0,32
```

However, if you want to use all the possibilities, try the "full version" of the command. Let's add that the value for the loop location must be between -128 and 127, so it will depend heavily on the file size and playback speed.

We said that you can change the volume during playback. Similarly, you can modify the pitch of the sound. The next command named SETPERIOD may be used for this purpose. Enter the sound number and the pitch, for example:

```
SetPeriod 1,2
```

Sound samples can be saved in different ways, therefore the value of the second argument should be determined based on the tests performed on specific files.

Using the above commands involves loading the entire contents of the file into memory. This will not always be possible due to the limitations of the maximum amount of RAM, in addition, long files saved on the disk could be played in parts.

The LOADSOUND command allows you to load files with a maximum size of 128 kilobytes, so today it is not an impressive value.

To listen to a longer soundtrack you can use larger files loaded from a hard disk or a CD. However, you must load them in parts, and thanks to this the playback is also possible on computers equipped with an average amount of free memory.

Such an operation is possible using the DISKPLAY command. We use it similar to SOUND, but the first argument should be replaced with the file name. The correct order is below:

- file name, preferably with full path,
- channel mask,
- channel volume.

The mask and volume should be set identically as before, therefore the number of arguments will change depending on the number of audio channels used.

The file name should be saved along with the path, but it is not absolutely necessary. The program will work according to the rules of AmigaDOS, so after typing file name without the path, the item will be searched in the current directory - the one in which your program is saved. You can also add the name of the directory, for example:

```
DiskPlay „data/audio3.iff”,3,64,64
```

This line will mean that the file is stored in an additional "data" directory and will be played on the first two channels at full volume. The rules for entering paths are typical for the entire Amiga operating system.

If you want to make the program independent of the directory where you save the finished program, use full paths, that is starting from the disk name, for example:

```
DiskPlay „Praca:pliki/data/audio3.iff”,3,64,64
```

Playing music from the disk is not completely independent of the amount of memory in the computer. The content is read in parts of specified sizes. The standard value is 1024 bytes (not kilobytes!), so it is the minimum portion suitable for each Amiga model. You can change it using the DISKBUFFER command. It is enough to provide a new size of the "buffer", i.e. for example:

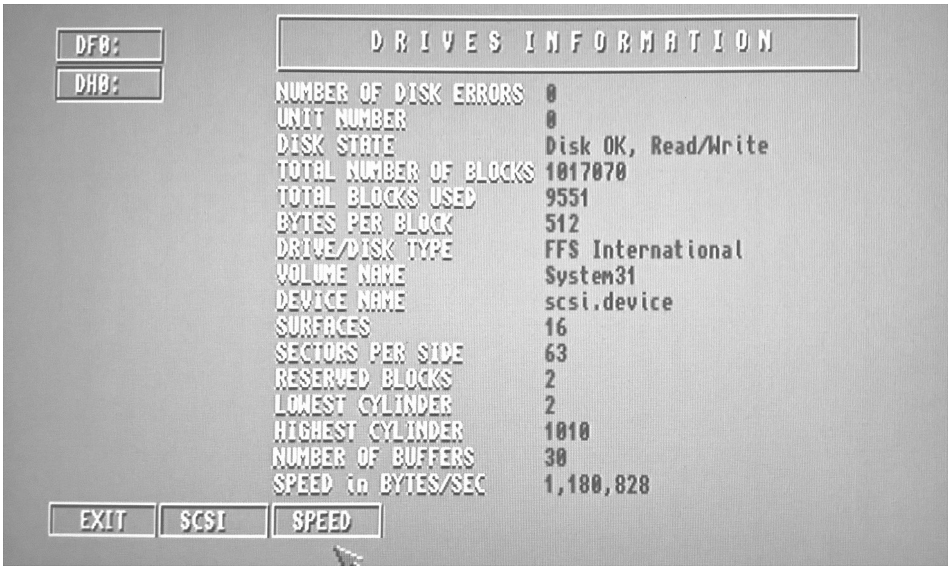
```
DiskBuffer 8192
```

Now it will be 8192 bytes, or 8 kilobytes. Remember that its size must be determined in relation not only the free memory and processor speed in the computer. Additionally - and maybe first of all - you need to check the speed of the medium from which the file will be read, and the quality of the recorded sound.

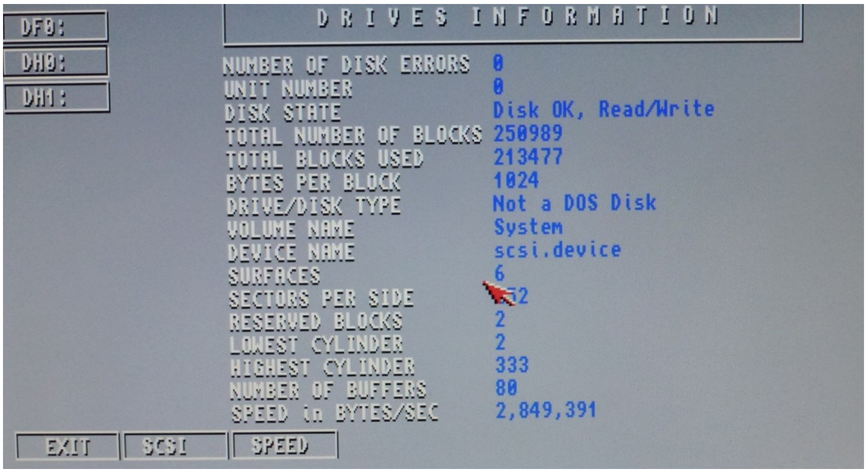
You can not read a long file from a floppy disk, so the CD/DVD, hard drive or memory card will be used. Both the mediums themselves and different types of devices allow get the various data read rates. In addition, issues related to the use of many file systems come.

Your program should be as universal as possible, so it can not act "on the edge of risk". You have to calculate the average speed of reading information in relation to the quality of the file, i.e. the frequency of the audio file.

For a better understanding, look at the sample results in the popular "SysInfo" program:



```
DF0:            D R I V E S   I N F O R M A T I O N
DH0:
                NUMBER OF DISK ERRORS    0
                UNIT NUMBER              0
                DISK STATE               Disk OK, Read/Write
                TOTAL NUMBER OF BLOCKS   1017070
                TOTAL BLOCKS USED        9551
                BYTES PER BLOCK          512
                DRIVE/DISK TYPE          FFS International
                VOLUME NAME              System31
                DEVICE NAME              scsi.device
                SURFACES                 16
                SECTORS PER SIDE         63
                RESERVED BLOCKS          2
                LOWEST CYLINDER          2
                HIGHEST CYLINDER         1010
                NUMBER OF BUFFERS        30
                SPEED in BYTES/SEC       1,180,828
EXIT   SCSI   SPEED
```



```
DF0:            D R I V E S   I N F O R M A T I O N
DH0:
DH1:            NUMBER OF DISK ERRORS    0
                UNIT NUMBER              0
                DISK STATE               Disk OK, Read/Write
                TOTAL NUMBER OF BLOCKS   250989
                TOTAL BLOCKS USED        213477
                BYTES PER BLOCK          1024
                DRIVE/DISK TYPE          Not a DOS Disk
                VOLUME NAME              System
                DEVICE NAME              scsi.device
                SURFACES                 6
                SECTORS PER SIDE         52
                RESERVED BLOCKS          2
                LOWEST CYLINDER          2
                HIGHEST CYLINDER         333
                NUMBER OF BUFFERS        80
                SPEED in BYTES/SEC       2,849,391
EXIT   SCSI   SPEED
```

The screens above come from different Amiga models, so you know how it might look like. What is the practical meaning? If you use too small a buffer, the sound will not play smoothly. Although, too high a value will cause a higher memory load.

As a consequence, the program will require more memory, which will be largely occupied for sound handling to the detriment of other elements.

Our suggestion is to check the situation on your computer, that is, determine the minimum value that allows you to reproduce the sound smoothly, and then increase this number by about 20-30% to establish a safe "margin of error".

However, it is best to share the program or just a part of it to several people who will check how it works on their Amiga configurations. Thanks to that, you will determine the best value.

Amiga also has the option of activating a sound filter, the so-called "low-pass". Its task is to extract a part of the frequency from the signal locating below the limit frequency. This is used in many programs and games. A characteristic feature of "turning on" or "turning off" this function is dimming the Power LED in the computer.

To enable the filter, use the line:

```
Filter On
```

or to disable it:

```
Filter Off
```

In addition to IFF 8SVX files playback, you can also use so-called "music modules", ie files containing music created realtime using sound samples - instruments. We will write more about it in the next section.

MUSIC

Blitz Basic allows you to play music not only recorded as sound samples, but also in the form of music modules. In short, the difference is in the fact that the module contains a list of music notes and additional parameters defining how to play sound samples on individual channels. These samples are treated as "instruments", so the music is created in real time - during playback. The IFF 8SVX file is just a sound sample, meaning a sound recorded from external audio source, which is played similarly to a CD. A single 8SVX file can be an instrument used in a music module.

In Blitz Basic, you can use modules in the "MOD" and "MED" format. The first one means the general format of 4-channel music, which began to be used in the "SoundTracker" program on the Amiga in the '80s. Please note that this file type can can has name created in two ways - the name may have an extension or the prefix "mod". In other words, it may look like this:

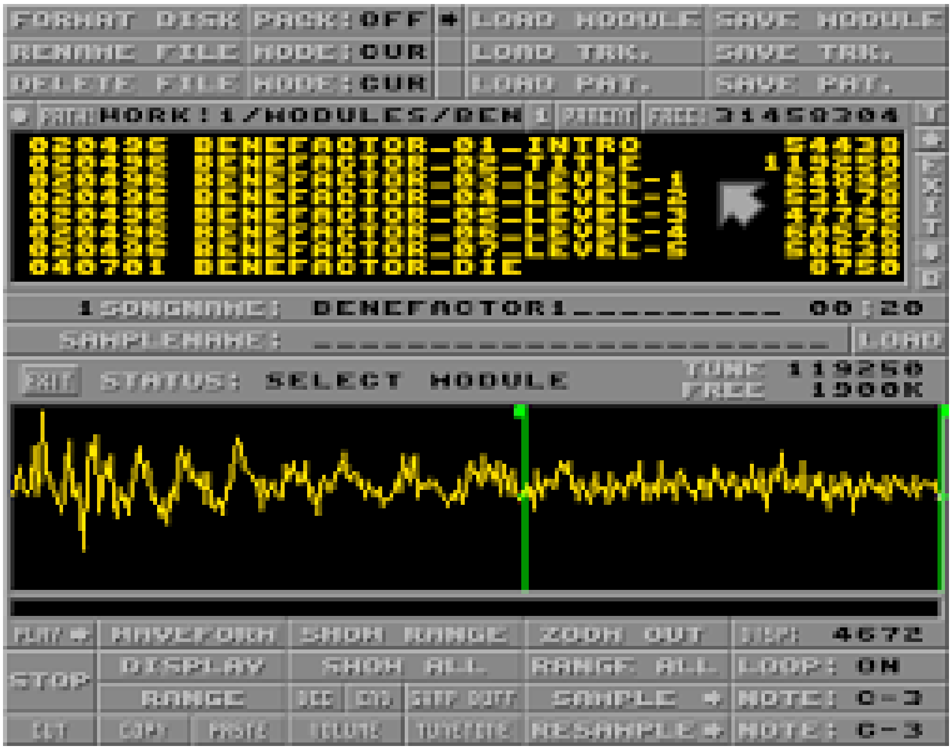
```
music.mod
```

or

```
mod.music
```

The first method is more widespread, because PCs once could not handle such a strange name as "mod." instead of ".mod" in '80s and even '90s. Hence, users changed the names to unify the items and enable seamless operation on different operating systems. However, the prefix was originally used, not an extension, and you can find the files with different names. The "ProTracker" program itself allows you to view files regardless of the name.

It looks similar to the following illustration:



```
PACK:OFF
MODE:CUR
MODE:CUR
PATH: WORK!1/MODULES/BEN    FREE: 31450304

020426  BENEFACTOR_01_INTRO            54430
020426  BENEFACTOR_02_TITLE           119250
020426  BENEFACTOR_03_LEVEL-1          [illegible]
020426  BENEFACTOR_04_LEVEL-2          52170
020426  BENEFACTOR_05_LEVEL-3          47726
020426  BENEFACTOR_06_LEVEL-4          [illegible]
020426  BENEFACTOR_07_LEVEL-5          60638
040701  BENEFACTOR_DIE                  8750

1 SONGNAME: BENEFACTOR1__________     00:20
SAMPLENAME: _________________________
STATUS: SELECT MODULE     TUNE 119250
                          FREE  1500K
SIZE: 4672
LOOP: ON
NOTE: C-3
NOTE: C-3
```

It is worth remembering these differences, because using the older software collections, we can run programs that do not allow you to change the settings for file names support. Then we may be forced to change them later, to use files in Blitz Basic.

To play a MOD module, we must use the LOADMODULE command. In this case, you only need to enter the module number and file name, for example:

```
LoadModule 1,"music.mod"
```

Of course, the above line only loads the file into memory. Next, start playback using the following line:

```
PlayModule 1
```

The number next to name of the command must be the same as previous - written along with the LOADMODULE word. Stopping playback will be performed after the STOPMODULE command has been executed, but this time we do not give any number. Only one module can be played simultaneously and it is interrupted immediately. That's why the whole line looks like this:

```
StopModule
```

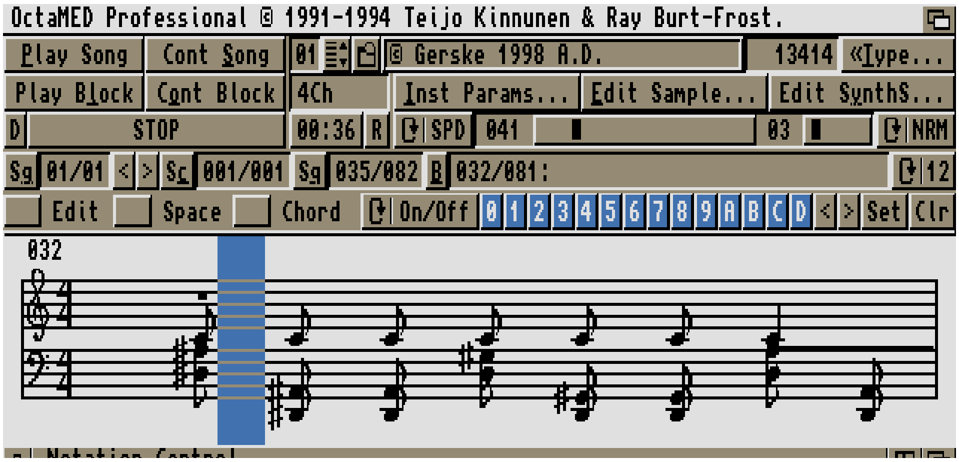
Remember that loading the file takes up memory, regardless of whether the module is playing or not. You can load more than one file at a time. When you turn on playback, the program may run a bit slower, because some time of the processor will be used to generate sound. However, this should not be a significant load, and you do not have to worry about it.

Finally you will need to free up memory for other operations. The module can be removed using the word FREE MODULE and this time you must also enter the module number. For example:

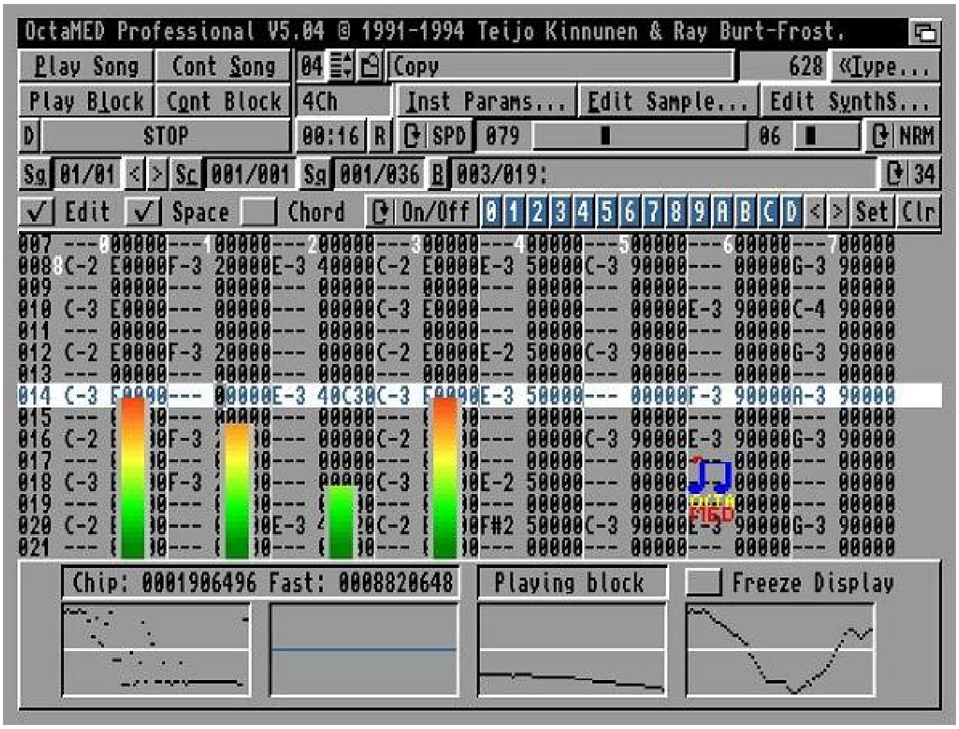
Free Module 1

After the time of the greatest popularity of 4-channel modules, Amiga users wanted to use more audio channels. In addition, it was postulated to create a program that uses the system user interface, as different monitors and display modes have already been used in various Amiga configurations. Unfortunately, the most famous "ProTracker", though it is a great editor, does not allow you to change the screen settings, which limits the usage. For these reasons, many similar programs have been created that operate on the same file format.

The new program called "MED" and its extended version called "OctaMED" were also created. Both use a new data format that allows you to save more information. The program also gives you possibility to create modules with more audio channels - at the beginning of 8 (hence the name beginning with "Octa"), and then up to 64 tracks. See the "OctaMED" interface, it is very different from "ProTracker":



or


OctaMED Professional V5.04 © 1991-1994 Teijo Kinnunen & Ray Burt-Frost.

```
007 ---000000---100000---200000---300000---400000---500000---600000---700000
0088C-2 E0000F-3 20000E-3 40000C-2 E0000E-3 50000C-3 90000--- 00000G-3 90000
009 --- 00000--- 00000--- 00000--- 00000--- 00000--- 00000--- 00000--- 00000
010 C-3 E0000--- 00000--- 00000C-3 E0000--- 00000--- 00000E-3 90000C-4 90000
011 --- 00000--- 00000--- 00000--- 00000--- 00000--- 00000--- 00000--- 00000
012 C-2 E0000F-3 20000--- 00000C-2 E0000E-2 50000C-3 90000--- 00000G-3 90000
013 --- 00000--- 00000--- 00000--- 00000--- 00000--- 00000--- 00000--- 00000
014 C-3 E0000--- 00000E-3 40C30C-3 E0000E-3 50000--- 00000F-3 90000A-3 90000
015 --- [illegible]0--- 00000--- 00000--- [illegible]0--- 00000--- 00000--- 00000--- 00000
016 C-2 [illegible]0F-3 [illegible]0--- 00000C-2 [illegible]0--- 00000C-3 90000E-3 90000G-3 90000
017 --- [illegible]0--- [illegible]0--- 00000--- [illegible]0--- 00000--- 00000--- 00000--- 00000
018 C-3 [illegible]0F-3 [illegible]0--- 00000C-3 [illegible]0E-2 50000--- 00000--- 00000--- 00000
019 --- [illegible]0--- [illegible]0--- [illegible]0--- [illegible]0--- 00000--- 00000--- 00000--- 00000
020 C-2 [illegible]0--- [illegible]0E-3 [illegible]0C-2 [illegible]0F#2 50000C-3 90000E-3 90000G-3 90000
021 --- [illegible]0--- [illegible]0--- [illegible]0--- [illegible]0--- 00000--- 00000--- 00000--- 00000
```

Chip: 0001906496 Fast: 0008820648

Modules stored in the "MED" format - it is determine by the name of the program - have the extension ".med" and this time you will probably not see the prefix. You can handle them in Blitz Basic, but you must use a different group of commands.

The basic method of operation is similar to the previous one, because the word LOADMEDMODULE should be used, but its arguments are similar to LOADMODULE. We give here the module number and its name, for example:

```
LoadMedModule 1,"music.med"
```

However, it allows you to get support for only 4-channel modules in this format. If you want to get more, you will be forced to use the functions of external system libraries. We will discuss it in detail later.

We play the ".med" file by entering the following line:

```
PlayMed
```

To stop music, simply enter:

```
StopMed
```

It is very simple, but it should be remembered that MED modules can cause a higher CPU load, because they allow to get effects that are not available for the MOD format. Therefore, you should remember to test the operation on different Amiga configurations. In addition, as we have already mentioned, using modules with 8 or more tracks requires special techniques.

As part of files with the extension ".med" you can find different file types and not everyone will be able to use in your program. Some of them you can try to save in a other way in the "OctaMED" program itself, but the conversion will not always be possible due to the functions used.

Blitz Basic in the field of MED format support offers more possibilities than just playing the music. First of all, you can start playing the module again without having to re-load the file. All you have to do is use the STARTMEDMODULE command, giving the number assigned to you in advance. Here's another example:

```
StartMedModule 1
```

At this point, we have reached a group of commands that do not have counterparts for the MOD format. The first is JUMPMED, which allows you to play music starting with specific part called "pattern". To understand this better, we have to say a few words about the internal structure of music modules.

Each file, regardless of whether it belongs to the MOD or MED format, contains a certain number of positions where music notes are stored, as well as corresponding instruments and other parameters. One "page" of such a entry is called "pattern". It may have different length, but it always has a similar structure. Patterns are arranged in a specific order, they can also be repeated and thus the whole module is played back.

In Blitz Basic, you can play a module from a specific pattern using the JUMPMED command. Just enter the number of the pattern, so the whole line looks like this:

```
JumpMed 5
```

Of course, the change applies to the currently playing module, so you can influence the music while listening. This can be useful in many cases, for example when writing dynamic games, where the soundtrack has to react to the action visible on the screen.

An example of such production can be a series of pinballs, from "Pinball Dreams" to the famous "Slam Tilt".

Please note that individual music parts are often switched or repeated in these games. This happens so quickly that it would not be possible without gameplay from hard disk, while these productions also work with floppy disks. The solution is just the appropriate structure of the music module, and then switching between "patterns". The playback does not require a fast processor, does not take up a lot of memory and allows you to save different parts of the music. Remember that different patterns do not have to be connected in any way. It all depends on the playback order, which is why it is a perfect format for this type of test.

Please also note that handling ".med" files requires the presence of a library named:

```
medplayer.library
```

Where to get it from? It is best to download the installation version of the "MED" or "OctaMED" program. You can also go to the Aminet website and download the file named "medplayer_lib.lha". Here is a direct link:

```
http://aminet.net/package/util/libs/medpla-
yer_lib.lha
```

When you play a MOD module, you can easily change its volume. All you need to do is use the SETMEDVOLUME command and enter a number between 0 and 64, for example:

```
SetMedVolume 32
```

The above line will set the volume at 50% of the possible range. In this case, you do not have to enter separate numbers for individual audio channels, because the operation is performed on all channels simultaneously. On the one hand, this limits the usage a little, on the other - thanks to this it is possible to create effects such as muting or amplifying when you want to smoothly start or end music playback.

However, if you want to modify the volume on different channels on an ongoing basis, you can do it with the word SETMEDMASK. You should give a mask, meaning a value of specific channels - as we discussed earlier. If you do not know how to do it go to the section entitled "Sound".

- Reading the volume

In some situations, you may need to get information about the current volume of the module being played. To do this, use the GETMEDVOLUME command. It allows you to read the volume of specific music channels, so its use is wider than only in relation to files with the extension ".mod" or ".med".

To find out what is the current volume of a given channel, enter the following line:

```
GetMedVolume 2
```

Number means the channel number. In result, you get the value corresponding to the current audio volume. This may be useful if you want to adjust the volume of the voice played in sequences, or you want to allow volume control by the user of your program. You can certainly know many uses of this function.

- Other audio formats

Remember that Amiga uses not only IFF 8SVX, MOD and MED formats. Many programs have been created to create music, but the usage requires the installation of additional software.

If you want to play other types of files, you should use external libraries, which can be found, among others, on this website:

Aminet.net

An example may be the archive "xmplay-lib.lha" containing files that allow to use the "FastTracker" program modules, i.e. with the extension ".xm". We will discuss system libraries later.

SPEECH

The first generation of Workbench contained a speech generator. This can be compared to modern speech synthesizers that can be found on the Internet. In subsequent editions of the Amiga operating system, the files responsible for speech were removed, along with a simple editor named "Say".

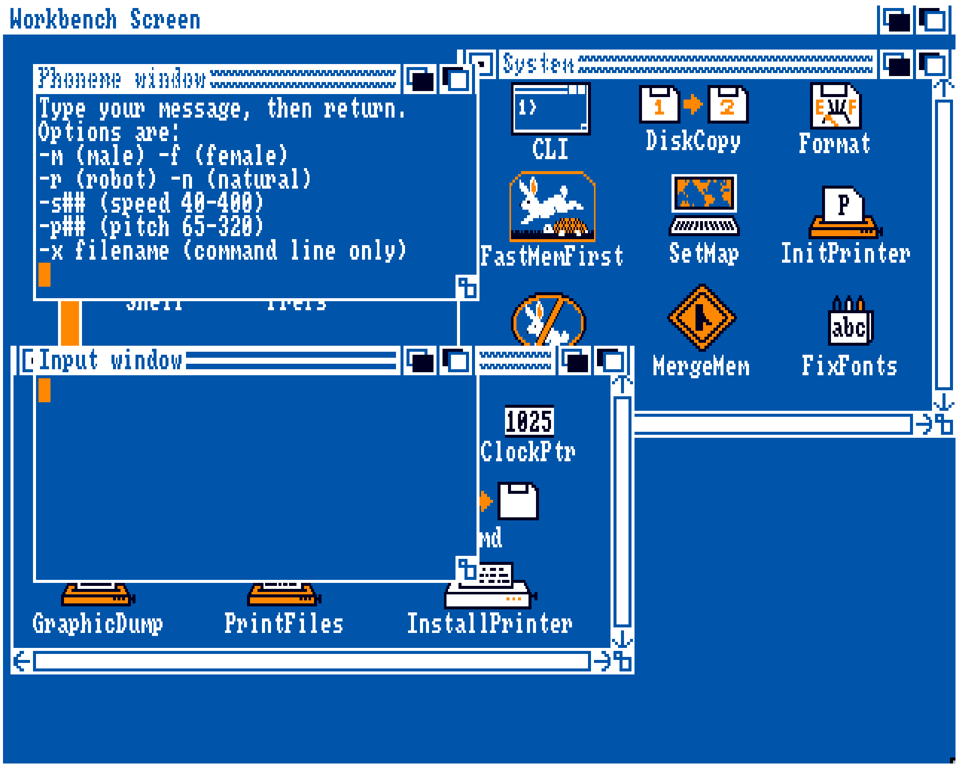
Converting text to voice is possible using a system device called "narrator" and it is still supported in Blitz Basic.

In order for the computer to "speak" the text, just use the SPEAK command, where we give the text variable. For example, the following lines:

```
a$=″Amiga i Atari″
Speak a$
```

cause that the computer will "say" words written in the "a$" variable. This is not the end of the possibilities, because we can change the sound of the "virtual voice instructor".

Please note that this is also possible with the program running even on Workbench version 1.3, i.e. on the Amiga 500. Then it looks like in the next picture:



```
Type your message, then return.
Options are:
-m (male) -f (female)
-r (robot) -n (natural)
-s## (speed 40-400)
-p## (pitch 65-320)
-x filename (command line only)
```

Similar changes can be obtained using the word SETVOICE. However, you must enter a few new arguments in the following order:

- rate,
- pitch,
- expression,
- sex,
- volume,
- frequency.

These parameters are difficult to understand without testing several different sets of functions. Therefore, it is best to change them by trial and error. You can also run Workbench 1.3 and the "Say" program to get an idea of a speech synthesizer.

Remember that this is not a product that can compete with modern products, but "spoken" words are clear and this feature can have many uses. The "narrator" device is rarely used today, but it is certainly worth exploring its capabilities. It is worth emphasizing that there is no problem with the use of various accents, and so many languages are supported, including Polish.

ADVANCED OPERATIONS

FILE HANDLING

In more complex programs you will need to read and write data not only using the "Open" and "Save" options, but also internally. This can be useful for maintaining temporary or other data when you want to keep the information on the disk without user control.

To access the file, you must indicate its name and path. In addition - as in many other cases discussed - the file must be assigned to a specific number. We do all this with one function called OPENFILE().

It should be used in the following way:

```
file3=OpenFile(1,”Work:data/data3”)
```

The above line will "open" a file named "data3" and assign it a number 1. More precisely, an access attempt will be made. If the operation was successful, you can check it by reading the value of the variable "file".

If everything is OK, logical TRUE (1) will be resulted, otherwise FALSE (0). The function behaves in a typical and legible manner.

Please pay attention to a few details. The file has no predefined purpose, so you can both read data and save information. This is a universal way, but sometimes it can cause problems. If the given file does not exist on the disk, then it will be created automatically.

What to do if you want to access a file from a CD or floppy disk, but without creating new data? You need to use another function called READFILE() in a similar way. This time it will not be possible to save data, but no other disk operations will be performed.

If the file does not exist, the function will result the value 0 (zero), or logical FALSE, and exit.

This can be very useful not only when there is not enough space on the floppy disk. Please note that you may need to read several files one by one from different directories or media. At the moment of transferring information about the lack of a file, you can create new procedure asking user to replace a floppy disk, for example a second diskette with your game.

In addition, in many cases the media is saved in such a way as to use almost the entire volume. If a new file would be always created, you could call the system message when the media is full. In addition, a user quickly exchanging floppies can do it during the operation and lead to the loss of the disk structure, i.e. the so-called "validation" failure. Typically, this results in information like this:

```
Disk not validated
```

All this makes it sometimes better to use the READFILE() function instead of the more general OPENFILE().

- Reading data

The information stored in the file can be read in several ways. First of all, we can use the FILEINPUT command. Next to it, enter the number of the "open" file, i.e. for example::

```
FileInput 0
```

This line will determine the usage of the file, in this case - data reading. However, it does not perform the operation, because it should be done with the EDIT$() function. As its argument, we give the number of characters we want to get. Of course, as with other functions, it must be assigned to a variable, for example:

```
dane[1]=Edit$(100)
```

We used an array where the element with index 1 will contain 100 characters. You can read more about the arrays in a separate section. Now note that this method can be useful in the context of various loops. Look at the more extensive program:

```
If ReadFile(1,"Work:data/data3")
     FileInput 1
          While NOT Eof(0)
                    For i=0 To 10
                          dane[i]=Edit$(100)

                    Next
          Wend
Endif
```

As you can see, the function can be directly placed in a line containing the word IF. However, in addition to two loops and conditional statements, we used the following entry:

```
EOF(0)
```

This is another function facilitating work, because it allows to localize the end of the file. The value used in the line with the word WHILE indicates that the end of the file (zero or logical FALSE) has not been reached. When the end of file is reached, the function will get the boolean value of TRUE (1) and the loop will also terminate.

The FILEINPUT command has a disadvantage. It works on the condition that the end-of-line character has not been placed in the file. To check this, you can use the CHR$() function. I write about it, among other things, under the section called "Operations on text variables".

When you know the contents of a file or you want to read the contents line by line, this is a very good choice. Otherwise, you may have problems with getting data.

That's why Blitz Basic allows you to read files in a different way using another function named INKEY$. The method of use is slightly different, because as the parameter you should specify only the number of characters you want to read, for example:

```
dane[1]=Inkey$(100)
```

The operation will always be performed in relation to the currently "open" file, so be careful when using it in loops.

In this case, you can also give the function without any argument, then only one character will be read by default. This may apply not only to information read from the disk. I write about it more under the section "File Handling".

- Saving data

Blitz Basic also allows you to open the file in the way so that only the writing data is possible. To make it possible, instead of the word FILEINPUT, enter the FILEOUTPUT command, by analogically adding the number of the "open" file. E.g:

```
a=OpenFile(1,″Work:data/data3″)
```

and later:

```
FileOutput 1
```

Now, to perform the operation of saving data in a file, you must use a completely different formula. We use ordinary commands that print information on the screen, that is:

```
PRINT
```

or

NPRINT

We wrote about it in the previous chapters. Remember that the "output" will be directed to the file indicated in the line containing the OPENFILE() function.

Another way to save data in a file is to use the WRITEFILE() function, which works analogously to the READFILE(). It automatically creates a new file and results the value 1, which is the logical TRUE, unless the operation fails.

Please note that in this case it is very easy to lose the contents of the file with the given name, because the function does not check whether the same item was previously saved on the disk.

The following I present an example of use the function as part of a conditional statement:

```
If WriteFile (1,"Work:data/data3”)
      FileOutput 1
            For i=0 To 10
                  NPrint dane[i]
            Next
Endif
```

Differences between the PRINT and NPRINT commands still apply. In other words, the second command will create a new line each time when on writing data. This is important when you create a file with entries executed repeatedly, as above.

- Ending work with the file

After the operation of reading or writing the information has been completed, the file should be "freed" from use, i.e. we have to close the transmission channel. This should be done using the following command:

```
CloseFile 1
```

Next to it, we simply give the file number. This does not mean interference with the content or deletion the item from the disk. We discuss this last function in the section "Removing files".

- Navigating the contents of the file

When we have the correct file created, we can read the data in a more sophisticated way. Using the previous commands calls the necessary functions, but we have no control over the current position within the content. It is possible using the additional command called FILESEEK.

As arguments, type the file number and position, but we do not know it yet. So at the beginning, you need to recognize the size of the file. We will obtain it using the LOF() function (Length-Of-File). It is enough, as usual, to assign it to a variable and enter the file number, for example:

```
plik=Lof(1)
```

Now we can create a program checking the position in the file. If you want to "jump" to a specific location, use the word FILESEEK as follows:

```
FileSeek 1,1024
```

The above line will set the position to 1024 bytes, which is the first kilobyte of file with number 1. If after the operation you want to check what is the new position, use the LOC() function (Location-Of-File). It should be used in a similar way to LOF(), but the value resulted will be the current position in the file. It can also be zero when the file has just been "opened" and we have not done anything on it.

Please note that taking into account the above functions and commands, you can easily write a program that adds data to a file that has already been created. You only need to check each time whether the file exists and then test the position.

You can also use the EXISTS() function. As an argument, enter the file name, so that we get a number corresponding to the size of the file.

At this point, you can have doubts about why to use this function, since it provides the same data as LOF(). However, these are only appearances, because the function informs in this way that the file is already "open" and is not empty. If there is a different result of the operation, then we get the value 0 (zero) and this is the difference from the previous function.

- Removing files

Sometimes you will need to delete an unnecessary file from the disk. This happens when using temporary data or when there is an error in the item. The operation of the removing file is very easy, it only requires entering the command name called KILLFILE. As an argument we give the name of the file. For example, like this:

```
KillFile „Work:data/data3”
```

For security reasons, it is always best to also type the full path. Otherwise, we can delete the file you need. If the entry contains a path, there is no possibility of confusion with another item on the disk.

MACROS

Macros (or macro definitions) are special parts of program that allow you to speed up its operation. They are usually written in Assembler and operate without the use of system libraries, so with the full speed of the processor.

In Blitz Basic, macros can be created in a similar way to procedures, but their purpose and way of calling are different. The program fragment must be placed between the words MACRO and END MACRO, additionally, you should enter the name in the first line. Here is an example:

```
Macro moje4
      a=5
      NPrint a
End Macro
```

In this way, we will create a macro called "moje4". Execution in the program is obtained after entering a line beginning with an exclamation mark. We can do this as follows:

```
NPrint „Hello”
!moje4
```

As you can see, simply enter the name of the macro after the exclamation mark. Its content must be placed earlier, that is, before the execution line. Note that this is not fundamentally different from handling other elements of the programming language.

The macros distinguish themselves from the way the program is run, because during compilation, the content placed between the MACRO and END MACRO commands will not be compiled, but stored in RAM memory, intented to the execution.

Thus, "jumps" to specific parts of the listing will not be performed, but the commands stored in the macrodefinition will be inserted in the place of macro name. In practice, our previous program will get the following form:

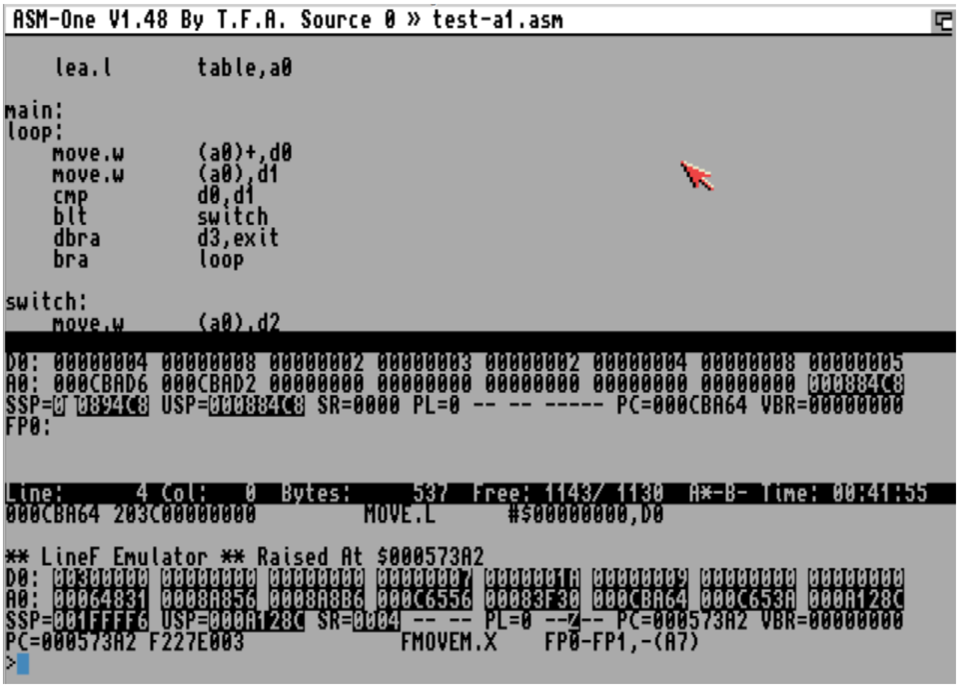
```
NPrint „Hello”

a=5
NPrint a
```

Thanks to this, the operation can be faster, which is especially important when writing larger programs dedicated to run on unexpanded Amiga models.

USING ASEMBLER

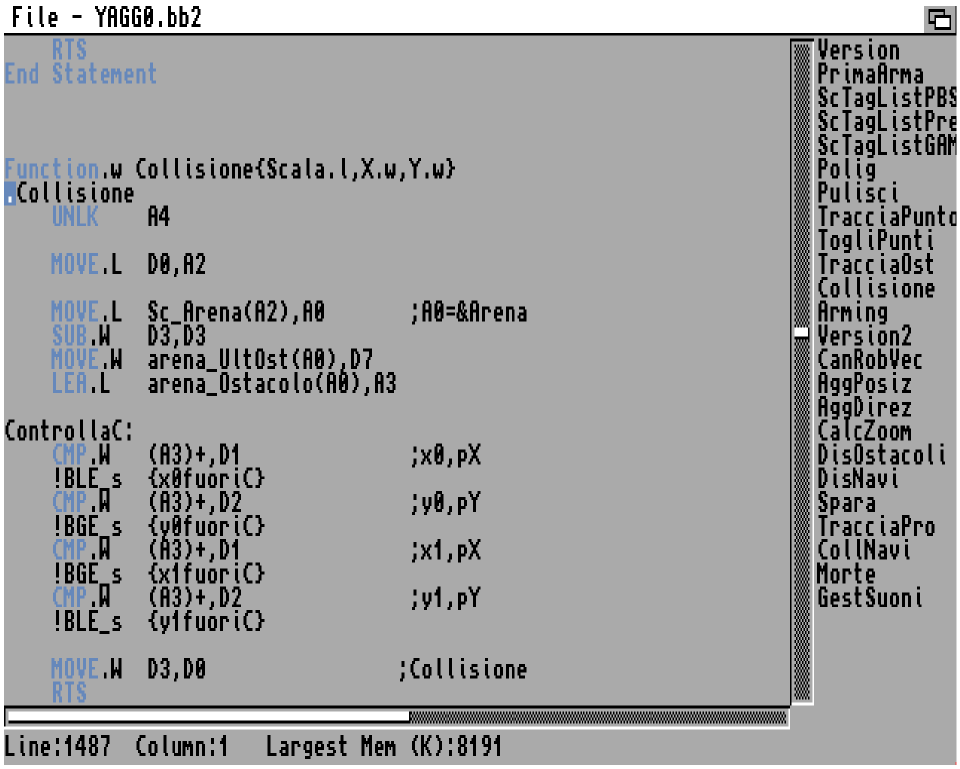
Blitz Basic allows you to use the machine code intended for the processor, i.e. the so-called Assembler. Writing parts of the program in this way is quite complicated, but thanks to this you can increase the operation speed and have a more direct impact on performance.

The easiest way to use Assembler capabilities is to place instructions as part of the procedure. You can enter individual commands similarly to editors such as "Asm-One". See how it can look in a separate program:



```
ASM-One V1.48 By T.F.A. Source 0 » test-a1.asm

    lea.l       table,a0

main:
loop:
    move.w      (a0)+,d0
    move.w      (a0),d1
    cmp         d0,d1
    blt         switch
    dbra        d3,exit
    bra         loop

switch:
    move.w      (a0),d2

D0: 00000004 00000008 00000002 00000003 00000002 00000004 00000008 00000005
A0: 000CBAD6 000CBAD2 00000000 00000000 00000000 00000000 00000000 000884C8
SSP=0 0894C8 USP=000884C8 SR=0000 PL=0 -- -- ----- PC=000CBA64 VBR=00000000
FP0:

Line:       4 Col:    0  Bytes:      537  Free: 1143/ 1130  A*-B- Time: 00:41:55
000CBA64 203C00000000          MOVE.L      #$00000000,D0

** LineF Emulator ** Raised At $000573A2
D0: 00300000 00000000 00000000 00000007 0000001A 00000009 00000000 00000000
A0: 00064831 0008A856 0008A8B6 000C6556 00083F30 000CBA64 000C653A 000A128C
SSP=001FFFF6 USP=000A128C SR=0004 -- -- PL=0 --Z-- PC=000573A2 VBR=00000000
PC=000573A2 F227E003            FMOVEM.X    FP0-FP1,-(A7)
>
```

and in the Blitz Basic editor:



```
    RTS
End Statement



Function.w Collisione{Scala.l,X.w,Y.w}
.Collisione
    UNLK     A4

    MOVE.L   D0,A2

    MOVE.L   Sc_Arena(A2),A0       ;A0=&Arena
    SUB.W    D3,D3
    MOVE.W   arena_UltOst(A0),D7
    LEA.L    arena_Ostacolo(A0),A3

ControllaC:
    CMP.W    (A3)+,D1              ;x0,pX
    !BLE_s   {x0fuoriC}
    CMP.W    (A3)+,D2              ;y0,pY
    !BGE_s   {y0fuoriC}
    CMP.W    (A3)+,D1              ;x1,pX
    !BGE_s   {x1fuoriC}
    CMP.W    (A3)+,D2              ;y1,pY
    !BLE_s   {y1fuoriC}

    MOVE.W   D3,D0                ;Collisione
    RTS
```

Here's another example:

```
Statement moje7{x,y}
     MOVE.l d0,a0
     MOVE a0,d1
     ADD.l d2,a0
     AsmExit
End Statement
```

Please note that before closing the structure, we put the word ASMEXIT, which results in "normal" mode of program operation.

You can also copy the results of operations to different registers of 68000 series processors. I wrote about them briefly in "Compilation and running the program" section. Registers are memory cells that are used to store information. We will write about it in more detail in the next book about advanced operations in both Blitz Basic and Assembler.

For now, you should know how to use registers in the program. Two commands GETREG and PUTREG can be used this purpose. The first allows you to place the result in a specific processor register. We give the name of the register and the value to be stored. For example, the following line:

```
GetReg d0,a
```

will cause the contents of variable "a" to be saved in the D0 register. You can use data registers from D0 to D7.

The PUTREG command operates inversely, that is, it transfers the value saved in the processor register to a variable that can be operated in the Blitz Basic program. It may look like this:

```
PutReg d1,c
```

This time, the value from the register D1 is saved in a variable named "c".

For now, the concept of registers, their use and the values assigned to them may not tell you much, but the most important is general way of use. Motorola 68000 series processors have many registers, and Blitz Basic has more commands related to Assembler support. We will deal with everything in detail later. However, if you have knowledge of writing programs in Assembler, the above information will allow you to use the machine code now.

SYSTEM INFORMATION

In Blitz Basic it is possible to directly test some of the computer components and operating system. This can be useful where you want to make the program operation depending on different processors or a specific Kickstart version. Sometimes it may be necessary for the program to work properly, but more often in this way you will be able to get better performance.

Using the PROCESSOR() function, we can see what CPU is installed in the Amiga. All you need to do is assign it to a variable, as we discussed it earlier, for example:

```
a=Processor
NPrint a
```

You do not have to give any arguments. In result, you will get one of the possible values, as below:

- 68000 CPU	**0**
- 68010 CPU	**1**
- 68020 CPU	**2**
- 68030 CPU	**3**
- 68040 CPU	**4**

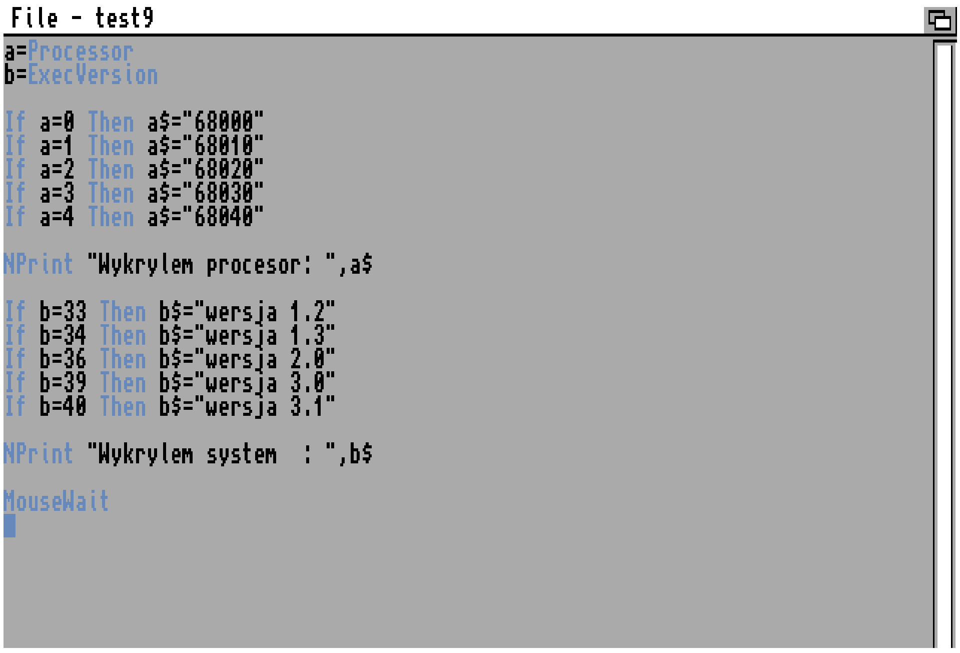
Also you can check the version of operating system, or more precisely the library "exec.library". The EXECVERSION() function is used for this. The usage is the same:

```
b=ExecVersion
NPrint a
```

but this time you can expect the following results:

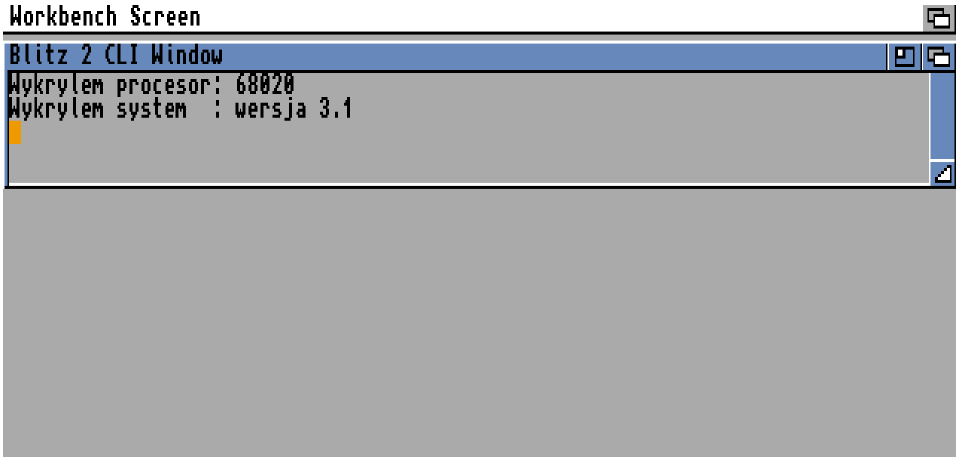
- system 1.2	**33**
- system 1.3	**34**
- system 2.0	**36**
- system 3.0	**39**
- system 3.1	**40**

Take a look at the example effect of the above functions. This is the listing presenting information about all of the following items:



```
a=Processor
b=ExecVersion

If a=0 Then a$="68000"
If a=1 Then a$="68010"
If a=2 Then a$="68020"
If a=3 Then a$="68030"
If a=4 Then a$="68040"

NPrint "Wykrylem procesor: ",a$

If b=33 Then b$="wersja 1.2"
If b=34 Then b$="wersja 1.3"
If b=36 Then b$="wersja 2.0"
If b=39 Then b$="wersja 3.0"
If b=40 Then b$="wersja 3.1"

NPrint "Wykrylem system  : ",b$

MouseWait
```

and this is the result of operation on Amiga 1200 equipped with Kickstart 3.1:



```
Wykrylem procesor: 68020
Wykrylem system  : wersja 3.1
```

Of course, your program does not have to display the values, it's much better to enter conditional statements or procedures that will work slightly different after detecting various pieces of hardware and software. In this way, the program can run more universally and faster in many cases.

ERROR HANDLING

ERRORS IN THE PROGRAM

If you write a program that allows user to use it freely, the user may cause an error. When the algorithms are not the simplest, there may also be an failure and your program will stop. Remember that even if you think it is unlikely, you are not able to foresee all possible situations.

- Error trapping

Please note that the program should not terminate, but inform the user of the error or run the appropriate procedure to eliminate the problem. Therefore, it is possible to "capture" problematic situations so that your program can continue to operate.

It is possible with a few commands, among which it is worth to know the one named SETERR. It can work when any error causing terminate of the program is called. To make this possible, at the beginning of the program you should place the structure SETERR ... END SETERR, for example:

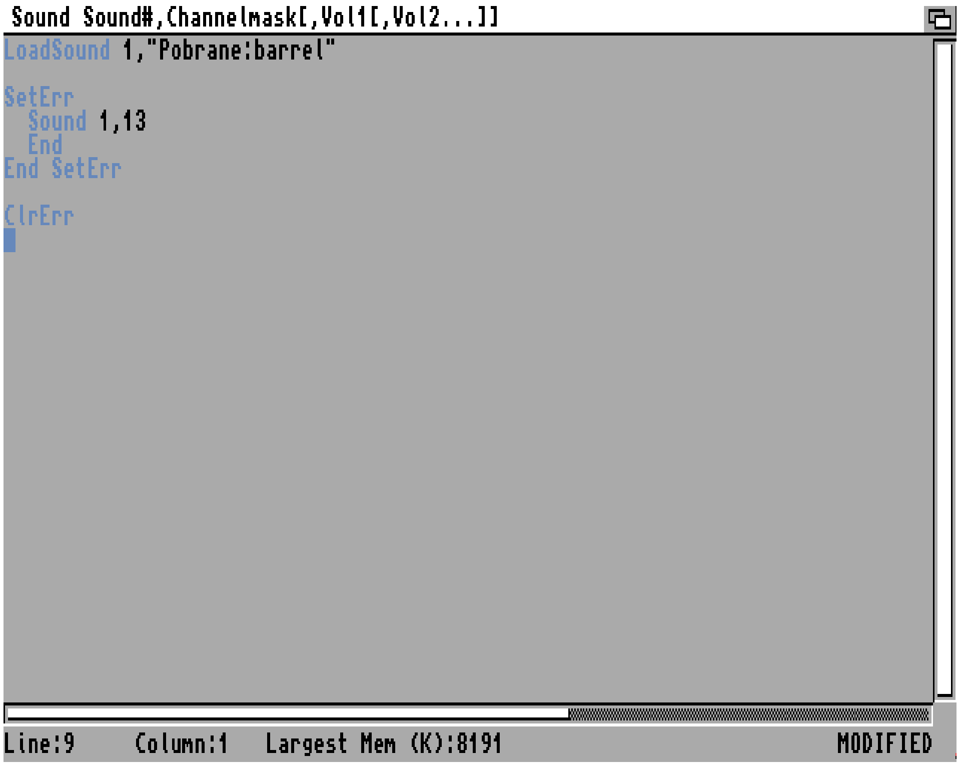
```
SetErr
     NPrint „Error!”
End
End SetErr
```

In the penultimate line, we put the END command, because in this situation the program should not go any further. Of course, instead of displaying a message - or the function that handles the error - you can "jump" to another part of the listing.

After completing the procedure, the error information is not automatically cleaned, so you should delete it. We do this with the CLRERR command. Just enter its name on a separate line.

If you do not do it, it will quickly turn out that after a single occurrence of the problem, it will be constantly duplicated, which of course we must avoid. Consequently, the program will not be able to run.

Look at a similar fragment of the program in the editor:



```
Sound Sound#,Channelmask[,Vol1[,Vol2...]]
LoadSound 1,"Pobrane:barrel"

SetErr
  Sound 1,13
  End
End SetErr

ClrErr

Line:9      Column:1    Largest Mem (K):8191                    MODIFIED
```

- Determining the type of error

A general "capture" of the problem is not always enough. In many cases, we have to change the operation of the program depending on the type of error that occurred. You can do this using the word SETINT, where you must give one of value from the following list:

```
 0 Empty transmission buffer (applies
   to serial port)

 1 Error reading or writing to the disk

 2 Aborting the program

 3 Problem related to one of the Amiga ports

 4 Interrupting the work of the Copper

11 Full transmission buffer (again applies
   to the serial port, as 0)

12 Error reading data from the floppy disk drive

13 Another external problem
```

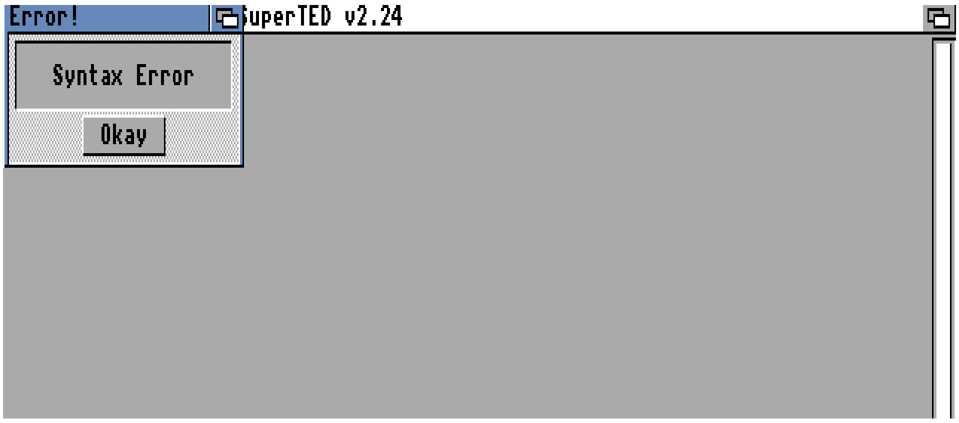
The SETINT statement alone is not enough, because the commands belonging to the error handling must be written in a similar structure as before, but this time the ending must contain the word END SETINT.

In the list of errors we have all the basic types. If you want to get more information or handle more detailed problems, you need to put additional variables in the program and then make the operate dependent on the SETINT ... END SETINT construct.

For example, if a serial port buffer overflow is recognized, the corresponding variable can record the amount of data transferred and if at least some of the files are complete, display the appropriate message for the user. Of course, there can be many uses, but the main rule always remains the same.

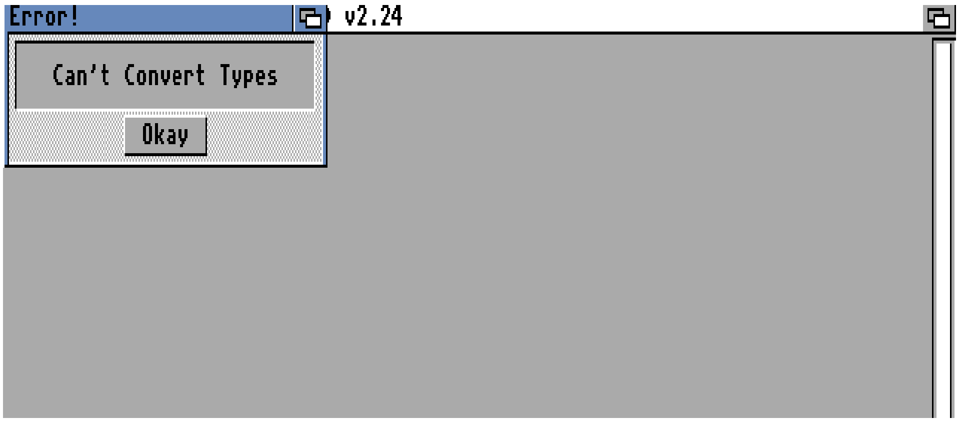
MESSAGES IN EDITOR

The "Ted" editor displays information about errors in the program. They have the form of a smaller window at the top of the screen. Here are two typical examples:



Syntax Error

and



Can't Convert Types

Select the "Okay" button to return to the listing editing and try to correct it. In this way you have a short description of the problem, but remember that it can have various type. Syntax errors are possible, ie incorrectly entered commands or variables, typos and other minor mistakes. Errors can be more serious and that is why it is not always easy to figure out the situation, and modifications can be difficult to implement.

Therefore, you should learn more about possible types of errors and explanations that appear on the screen. They are short, they do not have foreign language equivalents, so you can have doubts about the meaning. Thanks to the further part of this chapter, in most cases you will not have to think about the type of problem, but its solution.

SYNTAX ERRORS

The errors associated with syntax are easy to understand. These consist in incorrectly entering the name of the command, arguments or some other part. All this makes the line technically correct, but for example the number of arguments is too small or the wrong types have been determined, that is the number instead of the text string.

This type of error is signaled by following message:

```
Syntax Error
```

You have to check the exact content of the problematic line. You can also see another message, like this:

```
Garbage at End of Line
```

This means that there is no essential component in the line, such as a semicolon or comma separating individual command elements. When using different numeric values, you can use the wrong - too short or long line. In this case, you'll see an error:

```
Numeric Over Flow
```

Next problem may occur when using a line with the DATA command, but the values given are of a type not compatible with the variable type. Then Blitz Basic will signal this with the following information:

Bad Data

This error does not "say" too much, so check the entries in the DATA lines and compare to the rest of the program, which causes reading data and assigning them to a variable or variables.

Remember that the wrong syntax can be based on a small mistake like the typo, that may not be visible at first glance. This is very important, especially when you write a longer program and you try to quickly analyze the listing. Similar words may have different syntax, and even if you mistake only one character, the function may work in the wrong way.

LOGICAL ERRORS

The operation of the program depends on the commands, variables, various forms of algorithms and other elements. However, the program may work properly, but still not perform the intended functions. Therefore, theoretically, you can use it, but it is not fully functional.

Such problems are called logical errors. They are difficult to recognize and correct, because you have to do it yourself. The programming language editor itself will not indicate a problem, as it has no intelligence that can understand your intentions.

Similar situations occur more frequently, the more complex (not necessarily longer) is the program. You can eliminate them by following a few simple rules. The most popular techniques that reduce the number of logical errors include, among others:

- using understandable variable names, labels, procedures and other elements,

- avoiding using shortcuts that may be unclear or dependent on other parts of the program,

- creating the most simple and easy to analyze structures,

- using comments describing variables, goals and assumptions of the program, and separate the various parts.

These points are just a few basic schemes, but they will improve the readability of the program and make understanding the operation of algorithms much easier. Thanks to this, you can save time while testing the program, so you can achieve your goal faster and without complications.

ERRORS RELATED TO CONDITIONAL STATEMENTS

The next type of error is associated with conditional statements, that is the lines of type IF ... THEN or longer constructs IF ... ENDIF.

The most common problem is the lack of the word ENDIF when you began the line with the IF command. In this case, you'll see the following message:

```
If Without End If
```

Note that this can also happen if you use multiple conditions and put the word ENDIF in the wrong place. When you enter new conditions within a larger IF ... ENDIF structure, it may also happen that the number of corresponding commands is correct, but one is not always taken into account due to changes in your condition after the IF word.

You must correct the lines so that all valid lines are always executed or the position of the ENDIF command must be changed.

Your condition may also be too long. Blitz Basic allows you to write them in blocks with a maximum size of 32 kilobytes. This is quite a lot, but if you exceed this value, the following information will be displayed:

```
If Block too Large
```

In this case, divide the condition into several smaller parts or modify the program so that you do not have to check all the fragments every time. This can be achieved by operations such as:

- reducing the number of variables,
- inserting text variables to shorten the lines containing the condition,
- determining less detailed ranges for numerical variables,
- using the arrays,
- creating procedures that provide information in a different way than before.

and many others. I've provided some examples, but the actual modification will depend on the nature of your program. In extreme cases, you will have to change the entire algorithm used, but smaller corrections are usually sufficient.

ERRORS RELATED TO LOOPS

Loops are also associated with the use of specific words within a single structure that repeats a line. If you store too many FOR or NEXT commands in relation to the rest of the content, you will see the following information:

```
Duplicate For...Next Error
```

If the word FOR is missing, then Blitz Basic will display a message meaning that you can not "close" a loop that was not previously created. This is done in the following way:

```
Next without For
```

You can also make the reverse operation - when you use the FOR without the NEXT word. In this case, you will be informed by this message:

```
For Without Next
```

Similar messages exist for other types of loops, namely:

```
Until without Repeat
```

or

```
Repeat without Until
```

Of course, this applies to the absence of one of the given words within the REPEAT ... UNTIL structure.

The content of the loop may be too long, as in the case of conditional statements. You can find out about it when the error is displayed:

```
For...Next Block to Long
```

or

```
Repeat Block too large
```

The first concerns the FOR ... NEXT loop, the second - REPEAT ... UNTIL. In both cases, the maximum length is 30 kilobytes. You can rarely be in a situation where the place will be overflowed, but you should know about this limit.

The FOR ... NEXT loop may contain a variable of the wrong type. You will then see the information as below:

```
Bad Type for For...Next
```

It indicates that the variable located next to the word FOR must be a number, otherwise it will not be possible to perform subsequent loop executions.

You can use the ELSE command inside the WHILE ... WEND loop. If you enter it incorrectly, you'll see the following:

`Illegal Else in While Block`

Note that all loops expect specific types of variables and use individual words in an orderly manner. You can not change the order of commands freely, because the one always means the beginning and the other - the end of the loop.

ERRORS RELATED TO LABELS, PROCEDURES AND FUNCTIONS

If you use a large number of procedures, you can make mistakes by incorrectly entering arguments. If you enter too few of them, you'll see the following message:

```
Not Enough Parameters
```

A similar message will be shown in relation to the wrong number of command arguments or function parameters. In this situation, do not try to improve the line at random, but check the correct number of individual elements. Otherwise, you may lead to an unexpected program operation, although usually your program will just not be possible to run.

You can also use too many parameters or arguments in a line to execute the command. In this situation, another error will appear as below:

```
Too many parameters
```

Parameters may also have incorrect types. When you will use the variable not in the place where it should be or try to create a type with non-compatible content, an error will appear on the screen, which is the following message:

```
Illegal Parameter Type
```

Functions must have variables that will not overlap with those stored in the main program. Otherwise, the program will not work, because the following problem will be shown:

```
Duplicate parameter variable
```

Now, you need to correct the variables so that they indicate to items which do not have their equivalents outside the function.

When you create many procedures, it may happen that you try to enter one in the other, that is, nest the contents of one procedure in the framework previously created. Such entry is not allowed, so you'll see an error like this:

```
Can't Nest Procedures
```

The situation looks similar to the arrays, which - if you would like to make them available in the whole program - can not be declared as part of the procedure. You must move such an array from the main program or create a new parameter that transfers the content of the element with the specified index. If you do not do it, then will see another message:

```
Can't Dim Globals in Procedures
```

Remember that procedures must be called in a specific way. Commands related to labels and subprograms, such as GOTO or GOSUB, are not intended for this. Therefore, the attempt to run the procedure using them will finish with the following information:

```
Can't Goto/Gosub a Procedure
```

The next problem is related to the naming of specific parts of the program. When you try to call a procedure that does not exist, you'll see an error with the following content:

```
Procedure not found
```

This means that you have made a mistake in the name or indicated a part that was not created as a procedure. You'll get a similar message when you enter the name of the label that is not in the listing. The difference is the first part of information that looks like this:

```
Label not Found
```

As we said before, the names of procedures or labels can not overlap. You can not create two of the same items, because you would not know which commands in the program are correct. If you try to define two identical names, you will see information about the procedure:

```
Duplicate Procedure name
```

or label:

```
Duplicate Label
```

Please also note that the names of the various parts of the program can not be completely random. For example, they can not be identical to command names. You can forget about it, because there are many words you can enter into the listing.

Therefore, if the name is incorrect for various reasons, the following text will appear on the screen:

```
Illegal Procedure Call
```

or

```
Illegal Label Name
```

Analogously to the previous case, the first concerns the procedure and the second - labels.

Remember that each "open" function or procedure - as a complete structure - must be "closed" in an appropriate manner. These may be the words END STATEMENT, which we discussed in a separate section. When you forget about it, Blitz Basic will display next information as below:

```
Unterminated Procedure
```

There is one more important message regarding labels. Its content is like this:

```
Can't Access Label
```

This informs you that it is impossible to access the label. In practice, it usually appears in a situation where the name has not been defined before. You can also see this when calling the label will be impossible for other reasons.

ERRORS RELATED TO THE ARRAYS

Arrays are specific data sets and can easily be confused with ordinary variables. When you try to call an array that has not been defined, you will see the information as below:

```
Array not found
```

Usually this is related to the use of parentheses in relation to an element that occurs in the program, but it is not an array. Another reason may be the use of curly brackets instead of ordinary brackets. The above message may also be replaced by two others:

```
Array not yet Dim'd
```

or

```
Array not Dim'd
```

Their meaning is identical. On the other hand, you can forget that in the program you have already used an array with a specific name. If you want to create the same array for the second time, the following information will be printed:

```
Array already Dim'd
```

Each array has indexes, according to which different values of variables are saved. In Blitz Basic, you can not use complex indexes that could store more values, for example, separated by a comma. An attempt to enter such an index will result in the following error:

```
Illegal number of Dimensions
```

It means that an invalid (unsupported) index has been provided and needs to be corrected. At this point, we can say that this is the lack of handling, because there are programming languages that allow you to create more complex arrays.

Indexes can not also be variables. To say it more precisely: you can use variables when using arrays within a loop, but the index itself must always be a number. If you give a variable, you will be prompted with the message:

```
Can't Create Variable inside Dim
```

You can read more about the use of arrays in a separate section.

ERRORS RELATED TO MACROS

When creating a macros, you can also make at least a few mistakes. The first is to refer to a item that has not been defined. In that case, you'll see the following message:

```
Macro not Found
```

You can also try to create two macros with the same name. This will bring up the following information about the content:

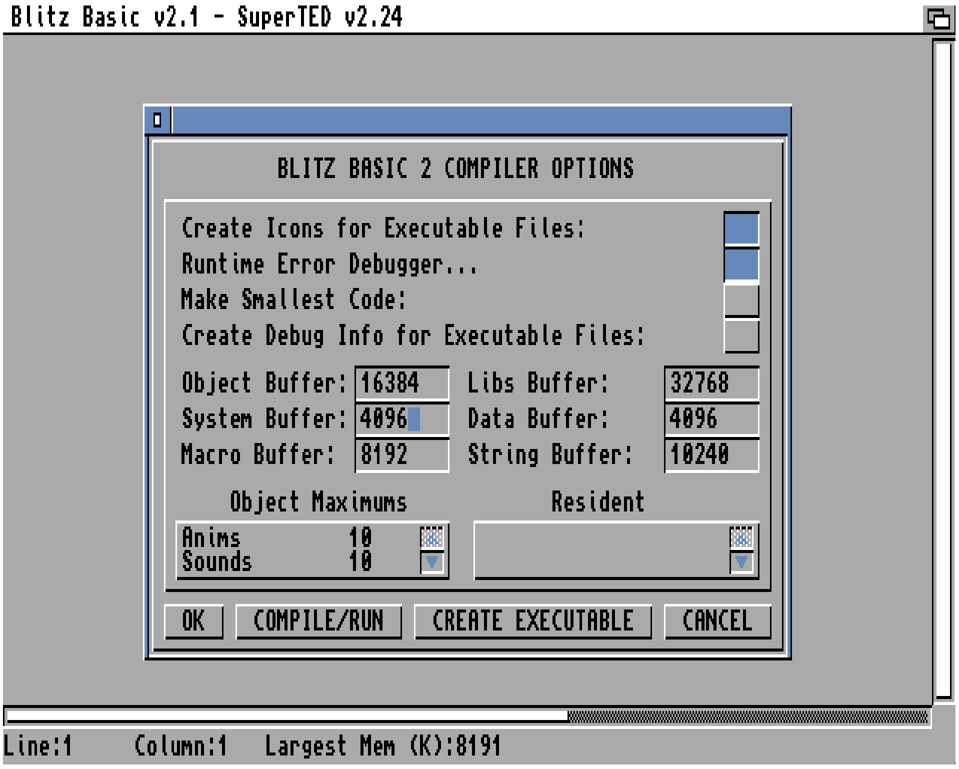
```
Macro already Defined
```

The structure may be too long, which will be signaled with the following inscription:

```
Macro too Big
```

Note that the size of the macro buffer can be changed in the configuration options. You have to call up the "Compiler" pull-down menu and then select the "Compiler Options..." option.

On the screen you will see a window where you can find several text fields. Change the value shown in the field named "Macro Buffer". As in our illustration:



```
Blitz Basic v2.1 - SuperTED v2.24

BLITZ BASIC 2 COMPILER OPTIONS

Create Icons for Executable Files:          ■
Runtime Error Debugger...                   ■
Make Smallest Code:                         □
Create Debug Info for Executable Files:     □

Object Buffer: 16384    Libs Buffer:    32768
System Buffer: 4096     Data Buffer:    4096
Macro Buffer:  8192     String Buffer:  10240

     Object Maximums            Resident
Anims        10
Sounds       10

OK   COMPILE/RUN   CREATE EXECUTABLE   CANCEL

Line:1     Column:1    Largest Mem (K):8191
```

If the buffer is overflowing after all, you may see another error like this:

`Macro Buffer Overflow`

It may also appear if your macro will be performed in an infinite loop, which should not happen.

Each macro must start with the word MACRO and end with ENDMACRO. If it does not, the program will not work and the following message will be displayed:

```
Macro without End Macro
```

This means that the listing lacks the "end", i.e. the ENDMACRO command. Remember that in this case the macros must be created in the main part of the program. When you try to do in another way, then Blitz Basic will print the information as below:

```
Can't create Macro inside Macro
```

The macros are quite complicated to use, while Blitz Basic does not contain a lot of errors messages related to them. Their usage is discussed in more detail in the chapter entitled "Advanced operations".

OTHER ERRORS

In the chapter "Error handling" we write about the SETINT ... END SETINT structure. If it is not entered correctly, you will see the following information:

```
End SetInt without SetInt
```

or

```
SetInt without End SetInt
```

The first means that the beginning of the structure, meaning the word SETINT, is missing. The second one points to a wrong ending. If you think all the commands have been entered, you probably made a mistake in a single character.

A similar situation may occur when creating the SETERR ... END SETERR construct. And the same can be shown when we will miss one of the required words:

```
End SetErr without SetErr
```

On the other hand, if you check the type of error and enter an incorrect number, that is an unsupported number indicating the problem, you can expect the following message:

```
Illegal Interrupt Number
```

For more information, see the section entitled "Determining the type of error."

An error may occur while reading the file. This is not a dangerous situation, but the data will not be loaded, so the program can not be executed. The following message will appear on the screen:

```
Error Reading File
```

NOTES
ON ERROR MESSAGES

Remember that mentioned error messages are just selected information you can see in the Blitz Basic editor. When a similar message appears, you should write down the words. A similar-looking message does not necessarily have to refer to the same element or the same situation in the program. During the work you should learn to correctly recognize errors and the names of commands related to them.

Logical errors are also very important. They can cause that the displayed information will direct you to other parts of the program than actually requiring corrections. Therefore, always start the analysis from checking how the algorithm is executed, and only then focus on specific information about the status of variables or function parameters.

ENDING

As in the previous book, the assumed volume does not allow me to discuss all the possibilities of Blitz Basic. Many topics have only been signaled, so they will be continued in the next book. I will deal with issues such as operations using direct support of Amiga chipset, using the ARexx scripting language, using more components of the operating system along with the creation of a complete user interface and many other topics.

I will also say more about the new version of the language called "AmiBlitz", as well as the preparation of programs for the AmigaOS 4 and MorphOS systems. There will be more extensive examples, and less explanation of the basic rules and operation. You will find this in the content of a separate series entitled "Programming for advanced".

I hope that my considerations will increase the interest in programming by Amiga users or people running famous emulators. Creating your own program or game always requires many hours of study and work, sometimes it brings disappointments, but above all brings a lot of satisfaction and broadens the horizons of our knowledge. Blitz Basic also exists for PCs, so the experience can be used in the field of other operating systems.

APPENDIX

INDEX: COMMANDS AND FUNCTIONS

Q

X

Y

Z

TECHNICAL INFORMATION

BLITZ BASIC

Author:
Mark Sibly

Release date:
1994

Operating system:
AmigaOS

Website:
www.blitzbasic.com

AMIGA.net.pl